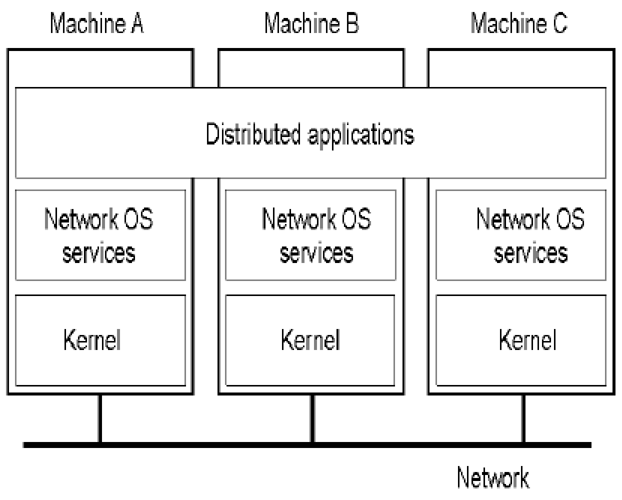
DISTRIBUTED SYSTEMS

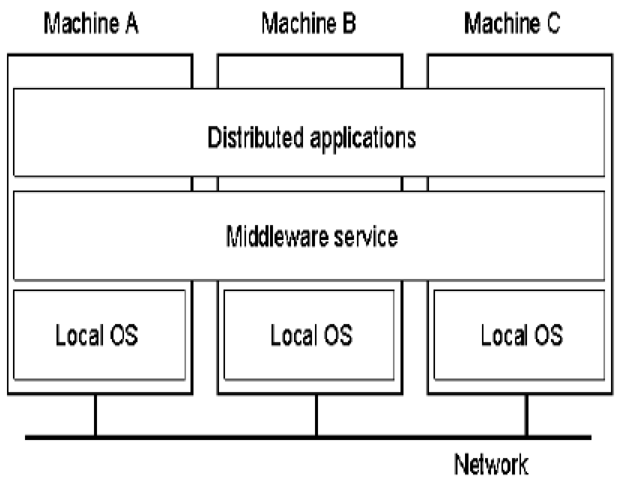
(written by Davide Giannubilo a.y. 23/24)

2. Modelling

2.1 The software architecture of a distributed system

- Network OS based
  - The network OS provides the communication services
  - Different machines may have different network OSes
  - Masking platform differences is up to the application programmer
- Middleware based
  - The middleware provides advanced communication coordination, and administration services
  - masks most of the platform differences





Middleware provides "*business-unaware*" services through a standard API, which raises the level of the communication activities of applications. Usually it provides:
Communication and coordination services

- Synchronous and asynchronous
- Point-to-point or multicast
- Masking differences in the network OS

Special application services

- Distributed transaction management, groupware and workflow services, messaging services, notification services, ...

Management services

- Naming, security, failure handling, ...

2.2 The run-time architecture

- **Client – Server**: the most common architectural style.
    - Servers provide a set of services through a well-defined API (they are passive)
    - Users are the clients who use those services
    - Communication is message based (or RPC)
- **Tiers**: multi-tiered client-server applications can be classified looking at the way such services are assigned to the different tiers, so we have a distributed application among these tiers.
  The services offered can be partitioned in 3 classes:
    - User interface services
    - Application services
    - Storage services
- **Service Oriented**: this architecture is built around the concepts of *service providers*, *service consumers*, *service brokers*.
    - Services represent loosely coupled units of functionality exported by the service providers
    - Brokers hold the description of available services to be searched by consumers which bind and invoke the services they need
    - *Orchestration* is the process of invoking a set of service in an ad-hoc workflow to satisfy the goal.

  (Examples: JXTA, Jini, Web Services, Open Grid Services Infrastructure)
  Web Service: "a software system designed to support interoperable machine-to-machine interaction over a network". Its interface is described WSDL (Web Service Description Language), and its operations are invoked through SOAP (protocol based on XML) which defines how the messages are exchanged.
  UDDI (Universal Description Discovery & Integration) describes the rules that allows web services to be exported and searched through a *registry*.
- **REST (REpresentational State Transfer)**: is "*a nice way to describe the web*" and "*a set of principles that define how Web standards are supposed to be used*".
  Key goals:
    - Scalability of component interactions
    - Generality of interfaces
    - Independent deployment of components
    - Intermediary components to reduce latency, enforce security and encapsulate legacy systems

  Main constraints:
    - Interactions are client-server and are stateless (state must be transferred from clients to servers)
    - The data within a response to a request must be implicitly or explicitly labelled as cacheable or non-cacheable (this is the key to provide scalability)
    - REST is layered (each component cannot "see" beyond the immediate layer with which they are interacting)
    - Clients must support code-on-demand (this is optional)
    - Components expose a uniform interface

The uniform interface exposed by components must satisfy 4 constraints:

1. Identification of resources, each resource must have an id (usually an URI) and everything that have an id is a valid resource (including a service)
2. Manipulation of resources through representations
    a. REST components communicate by transferring a representation of a resource in a format matching one of an evolving set of standard data types (e.g., XML), selected dynamically based on the capabilities or desires of the recipient and the nature of the resource
    b. Whether the representation is in the same format as the raw resource, or is derived from the resource, remains hidden behind the interface
    c. A representation consists of data and metadata describing the data
3. Self-descriptive messages
    a. Control data defines the purpose of a message between components, such as the action being requested or the meaning of a response
    b. It is also used to parameterize requests and override the default behaviour of some connecting elements (e.g., the cache behaviour)
4. Hypermedia as the engine of application state
    a. Clients move from a state to another each time process a new representation, usually linked to other representation through hypermedia links

- **Peer-to-peer**: in this architecture all components play the same role, so there is no distinction between client and server.
  Why do we need this?
    - Client-Server does not scale well due to centralization of service
    - The serve is a single point of failure
    - P2P leverages off the increased availability of broadband connectivity and processing power at the end-host to overcome such limitations

  P2P promotes the sharing of resources and services through direct exchange between peers like network bandwidth, data or storage space.
- **Object oriented**: the distributed components encapsulate a data structure providing an API to access and modify it, so each component is responsible for ensuring the data integrity of the data structure it encapsulates and the internal organization of such data structure is hidden to other components.
  It is a "peer-to-peer" model but it is also used to implement client-server applications.
  Components interact through RPC (Remote Procedure Call).
  Pros:
    - information hiding hides complexity in accessing/managing the shared data.
    - Encapsulation plus information hiding reduce the management complexity.
    - Objects are easy to reuse among different applications.
    - Legacy components can be wrapped within objects and easily integrated in new applications.
- **Data-centred**: components communicate through a common (usually passive) repository (Data can be added to the repository or taken (moved or copied) from it). Communication with the repository is (usually) through RPC. Access to the repository is (usually)

synchronized.
Linda and tuple spaces: data sharing model mostly used for parallel computation and communication is persistent, implicit, content-based, generative.
  - Data is contained in ordered sequences of typed fields (tuples)
  - Tuples are stored in a persistent, global shared space (tuple space)
  - Operations:
    - *out(t)* – write t
    - *rd(p)* – returns a copy of a tuple matching the pattern p, if it exists; if many matching exist, one is chosen non-deterministically
    - *in(p)* – like *rd(p)* but withdraws the matching tuple from the tuple space
    - *eval(a)* – some implementations provide this operation which inserts the tuple generated by the execution of a process *a*

  There are some variants like those *asynchronous*.
  Some issue with this architecture:
  - The tuple space model is not easily scaled on a wide-area network
  - The model is only proactive (processes explicitly request a tuple query)
  - Provide only client access to a server holding the tuple space
- **Event-based**: components collaborate by exchanging information about occurrent events. Communication is:
  - Purely message based – asynchronous – multicast – implicit – anonymous
- **Mobile code**: a style based on the ability of relocating the components of a distributed application at run-time. Different models depending on the original and final location of resources, know-how (the code) and computational components (including the state of execution).
  - Client-Server – send a request and the server does everything
  - Remote evaluation – send the code and the server does the computation
  - Code on demand – ask to a server for the code
  - Mobile agent – ask to a server to use its resources

  Mobile code technologies:
  - *Strong mobility* is the ability of a system to allow migration of both the code and the execution state of an executing unit to a different computational environment (very few systems provide it).
  - *Weak mobility* is the ability of a system to allow code movement across different computational environments (provided by several systems like Java, the Web, .Net).

  Pros:
  - the ability to move pieces of code (or entire components) at run-time provides a great flexibility to programmers (new services added, new functionalities can be added to existing components, etc.).

  Cons:
  - securing mobile code applications is a mess.
- **CREST (Computational REST)**: it joins together the concepts of REST and mobile code. Instead of "*representations*" interacting, parties exchange "*computations*".
  Axioms:
  - A resource is a locus of computations named by an URL.

    - The representation of a computation is an “expression” plus metadata to describe it
    - All computations are context-free.
    - Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged.
    - The presence of intermediaries is promoted.

2.3 The interaction model

Traditional programs can be described in terms of the algorithm they implement. Steps are strictly sequential and (usually) process execution speed influence performance, only.

Distributed systems are composed of many processes, which interact in complex ways. The behaviour of a distributed system is described by a distributed algorithm (a definition of the steps taken by each process, *including the transmission of messages between them*).

The behaviour of a distributed system is influenced by several factors:

- The rate at which each process proceeds
- The performance of the communication channels
- The different clock drift rates

To formally analyse the behaviour of a distributed system we must distinguish (at least in principle) between:

- Synchronous distributed systems
    - The time to execute each step of a process has known lower and upper bounds
    - Each message transmitted over a channel is received within a known bounded time
    - Each process has a local clock whose drift rate from real time has a known bound
- Asynchronous distributed systems
    - There are no bounds for process execution speeds, message transmission delays, clock drift rates

(Any solution that is valid for an asynchronous distributed system is also valid for a synchronous one, but not the vice versa)

2.4 The failure model

The failure model defines the ways in which failure may occur to provide a better understanding of the effects of failures.
We distinguish between:

- Omission failures
    - Processes: fail stop (other processes may detect certainly the failure) vs. Crash
    - Channels: send omission, channel omission, receive omission
- Byzantine (or arbitrary) failures
    - Processes: may omit intended processing steps or add more
    - Channels: message content may be corrupted, non-existent messages may be delivered, or real messages may be delivered more than once
- Timing failures (apply to synchronous systems, only)
    - Occur when one of the time limits defined for the system is violated

3. COMMUNICATION

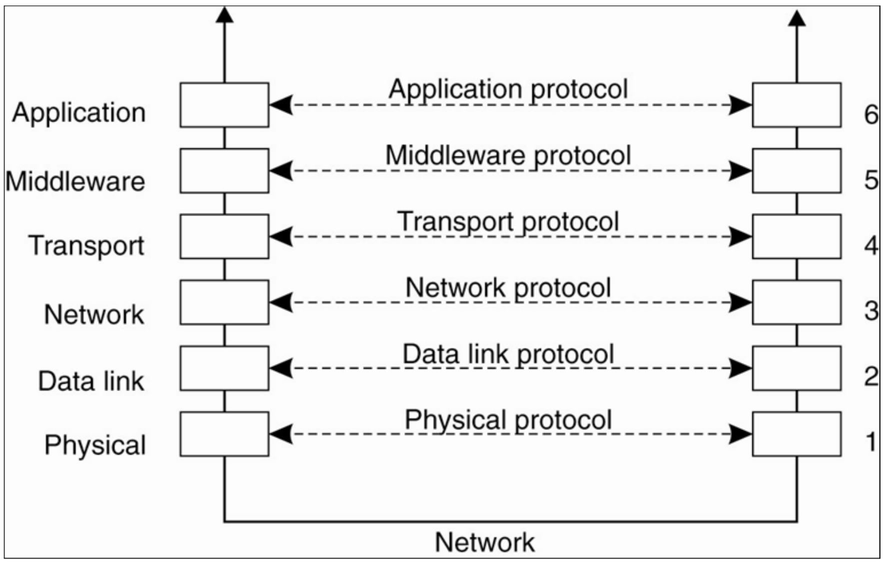
3.1 Fundamentals

The OSI model:

- Low layers
  - Physical layer (how bits are transmitted between 2 nodes directly connected)
  - Data Link layer (how a series of bits is packed into a frame to allow for error and flow control)
  - Network layer (how packets in a network of computers are to be routed)
- Transport layer (how data is transmitted among two nodes, offering a service independent from the lower layers): it provides the actual communication facilities for most distributed systems.
  - TCP (connection-oriented, reliable, stream-oriented communication)
  - UDP (unreliable datagram communication)
- Higher level layers
  - Session layer
  - Presentation layer
  - Application layer

Middleware includes common services and protocols that can be used by many different applications.

- (Un)marshalling of data, necessary for integrated systems
- Naming protocols, to allow easy sharing of resources
- Security protocols for secure communication
- Scaling mechanisms, such as for replication and caching



It offers different form of communication:

- Transient vs persistent
- Synchronous vs asynchronous

They can be combined like:

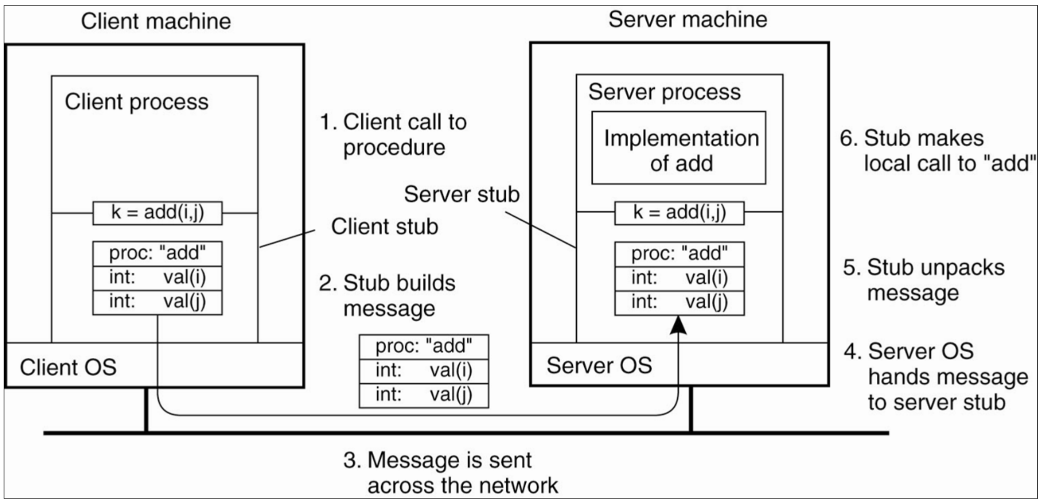
- Transient communication with synchronization after processing
- Persistent communication with synchronization at request submission

3.2 Remote Procedure Call (not persistent, in general)

When we make a local call to a function, we can pass some parameters in different ways:

- By value
- By reference
- By copy/restore

From local to remote procedure call (when passing parameters by value)



When passing some parameters, we could have 2 problems:

1. Structured data (like struct) must be ultimately flattened in a byte stream (called ***serialization***)
2. Hosts may use different data representations (like little endian vs big endian) and a proper conversion is needed (called ***marshalling***)

Middleware provides automated support:

- The marshalling and serialization code is automatically generated from and becomes part of the stubs. This is enabled by:
  - A language/platform independent representation of the procedure’s signature, written using an ***Interface Definition Language*** (IDL)
  - A data representation format to be used during communication

The **IDL** raises the level of abstractions of the service definition, in fact it separates the *service interface* from its *implementation* and the language comes with “*mappings*” onto target languages.

Advantages:

- Enables the definition of services in a language-independent fashion

- Being defined formally, an IDL description can be used to automatically generate the service interface code in the target language

How to pass a parameter by *reference*?
Many languages do not provide a notion of reference, but only of pointer. Often, this feature is simply not supported (as in Sun's solution).

Sun Microsystems' RPC (also called Open Network Computing RPC) is the *de facto* standard over the Internet.

- At the core of NFS, and many other (Unix) services
- Data format specified by XDR (eXternal Data Representation)
- Transport can use either TCP or UDP
- Parameter passing:
    - Only pass by copy is allowed (no pointers). Only one input and one output parameter
- Provision for DES security

The Distributed Computing Environment (DCE) is a set of specifications and a reference implementation.

Main problem of RPC: find out which server provides a given service.

- Sun's solution: they introduce a daemon process (`portmap`) that binds calls and server/ports:
    - The server picks an available port and tells it to `portmap`, along with the service identifier
    - Clients contact a given `portmap` and request the port necessary to establish communication
    - `portmap` provides its services only to local clients, so they must know in advance where the service resides. However, a client can multicast a query to multiple daemons.
- DCE's solution: it works like `portmap`, but we have a directory server that enables location transparency
    - Client need not know in advance where the service is: they only need to know where the directory service is.
    - In DCE, the directory service can be distributed (to improve scalability over many servers)

Another problem: server processes may remain active even in absence of requests, wasting resources (*dynamic activation*).

- Introduce another local server daemon that:
    - Forks the process to serve the request
    - Redirects the request if the process is already active
    - Clearly, the first request is served less efficiently

  In Sun RPC is `inetd` daemon.

Lightweight RPC: it uses local facilities (the kernel) to pass messages. Using conventional RPC would lead to wasted resources, no need for TCP/UDP on a single machine. (Uses less threads/processes)

Asynchronous RPC: RPC preserves the usual call behaviour, but the caller is suspended until the call is done. Potentially wastes client resources, like if no return value is expected and, in general, concurrency could be increased.

Batched RPC vs. Queued RPC

- Sun RPC includes the ability to perform *batched* RPC
  - RPCs that do not require a result are buffered on the client
  - They are sent all together when a non-batched call is requested
- A similar concept can be used to deal with mobility
  - If a mobile host is disconnected between sending the request and receiving the reply, the server periodically tries to contact the mobile host and deliver the reply
  - Requests and replies can come through different channels

3.3 Remote method invocation

The aim is to obtain the advantages of OOP also in the distributed setting.
In RPC, the IDL separates the interface from the implementation to handle platform/language heterogeneity. Such separation is one of the basic OO principles.

In practice *remote method invocation* works with:

- Java RMI
  - Single language/platform (Java and the Java Virtual Machine)
  - Easily supports passing parameters by reference or "by value" even in case of complex objects
  - Supports code on demand
- OMG CORBA
  - Multilanguage (by IDL) / multiplatform (by reference)
  - Supports passing parameters by reference or by value

3.4 Message oriented communication

- Centred around the (simpler) notion of one-way message/event
- Often supporting multi-point interaction
- Brings more decoupling among components

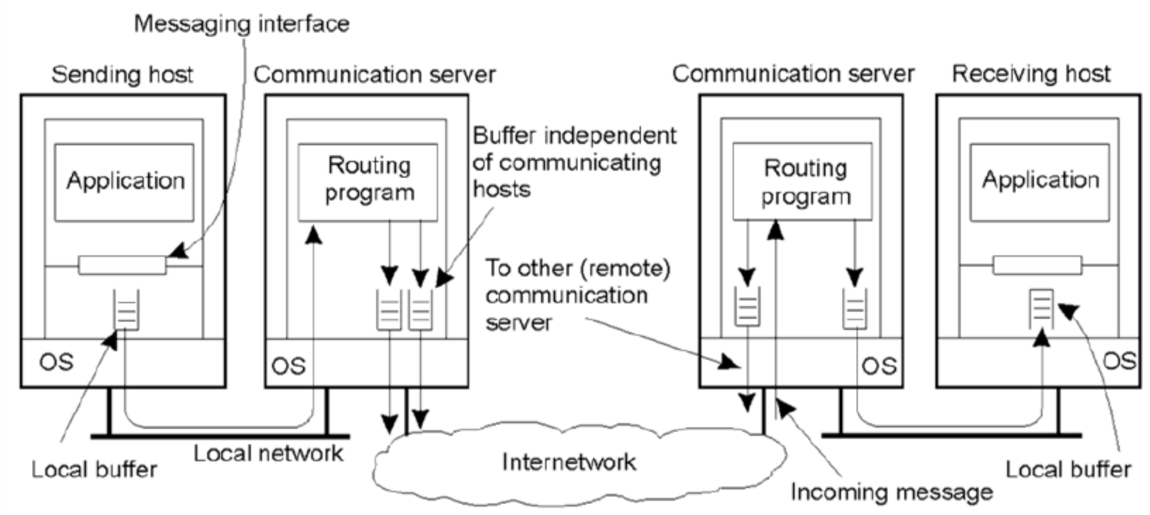
Types of communication:

- Synchronous vs. asynchronous
  - Synchronous: the sender is blocked until the recipient has stored (or received, or processed) the message
  - Asynchronous: the sender continues immediately after sending the message
- Transient vs. persistent
  - Transient: sender and receiver must both be running for the message to be delivered

- Persistent: the message is stored in the communication system until it can be delivered

The most straightforward form of message-oriented communication is *message passing*. Typically, directly mapped on/provided by the underlying network OS functionality (e.g., socket).
A (kind of) middleware provides another form of message passing called MPI.

Message queuing and *publish/subscribe* are two different models provided at the middleware layer by several "communication servers" and through what is nowadays called an *overlay network*.



Protocols and communication services:

- Unicast TCP and UDP (and multicast IP) are well known network protocols

How can a programmer take advantage of such protocols? Using Berkeley sockets.

- Sockets provide a common abstraction for inter-process communication.
  They allow connection-oriented (TCP) or connectionless (UDP) communication.

How a stream socket works? (TCP)

1. The server accepts connection on a port
2. The client connects to the server
3. Each (connected) socket is uniquely identified by 4 numbers: IP address of the server, its "incoming" port, IP address of the client, its "outgoing" port

How a datagram socket works? (UDP)

1. Client and server use the same approach to send and receive datagrams
2. Both create a socket bound to a port and use it to send and receive datagrams
3. There is no connection, and the same socket can be used to send (receive) datagrams to (from) multiple hosts

Multicast socket

IP multicast is a network protocol to efficiently deliver UDP datagrams to multiple recipients. The Internet Protocol reserve a class D address space, from 224.0.0.0 to 239.255.255.255, to multicast

groups.
The socket API for multicast communication is like that one for datagram communication:

- Component interested in receiving multicast datagrams addressed to a specific group must *join* the group (using the `setsockopt` call)
  - Groups are open: it is not necessary to be a member of a group in order to send datagrams to the group
- As usual it is also necessary to specify a port
  - It is used by the OS to decide which process on the local machine to route packets to

(Most routers are configured to not route multicast packets outside the LAN)

Message Passing Interface (MPI)

There are some limitations with sockets. They are low level, protocol independent, in high performance networks (e.g., clusters of computers) we need higher level primitives for asynchronous, transient communication, etc.

MPI is the answer.

- Communication takes place within a known group of processes
- Each process within a group is assigned a local id
  - The pair (`groupID`, `processID)` represents a source or destination address
- Messages can also be sent in broadcast to the entire group (`MPI_Bcast`, `MPI_Reduce`, `MPI_Scatter`, `MPI_Gather`)
- No support for fault tolerance (crashes are supposed to be fatal)

Message queuing

It is a point-to-point persistent asynchronous communication.

- Typically guarantee only eventual insertion of the message in the recipient queue (no guarantee about the recipient's behaviour)
- Communication is decoupled in time and space
- Can be regarded as a generalization of the e-mail

Also, client-server architectures have server's queues. In fact, clients send requests to the server's queue and it, asynchronously, fetches request, processes them, and returns results in the clients' queues (clients need not remain connected).

Some issues:

- Queues are identified by symbolic names. Need for a lookup service, possibly distributed, to convert queue-level addresses in network addresses
- Queues are manipulated by queue managers. Local and/or remote, acting as relays aka applicative routers. They are often organized in an overlay network:
  - Messages are routed by using application-level criteria, and by relying on a partial knowledge of the network
  - Improves fault tolerance

    - Message brokers provide application-level gateways supporting message conversion

Publish – Subscribe

- Application components can *publish* asynchronous *event notifications* (just simply message), and/or declare their interest in event classes by issuing a *subscription*
- Subscriptions are collected by an *event dispatcher* component, responsible for routing events to all matching subscribers. It can be centralized or distributed.
- Communication is
    - Transiently asynchronous
    - Implicit
    - Multipoint
- High degree of decoupling among components
    - Easy to add or remove them

The expressiveness of the subscription language allows one to distinguish between:

- *Subject-based* (or topic-based)
    - The set of subjects is determined a priori
    - Analogous to multicast (e.g., subscribe to all events about "Distributed Systems")
- *Content-based*
    - Subscriptions contain expressions (event filters) that allow clients to filter events based on their content
    - The set of filters is determined by client subscriptions
    - A single event may match multiple subscriptions (e.g., subscribe to all events about a "Distributed System" class with date greater than 16.11.2004 and held in classroom D04)

The event dispatcher oversees collecting subscriptions and routing event notifications based on such subscriptions.

- Centralized – a single component collects subscriptions and forward messages to subscribers
- Distributed – a set of *message brokers* organized in an *overlay network* cooperate to collect subscriptions and route messages.
  The topology and the routing strategy adopted may vary.

Acyclic graph (network)

- Message forwarding
    - Every broker stores only subscriptions coming from directly connected clients
    - Messages are forwarded from broker to broker and delivered to clients only if they are subscribed
- Subscription forwarding
    - Every broker forward subscriptions to the others
    - Subscriptions are never sent twice over the same link
    - Messages follow the routes laid by subscriptions

Each time a broker receives a message it must match it against the list of received filters to determine the list of recipients.
The efficiency of this process may vary, depending on the complexity of the subscription language, but also on the forwarding algorithm chosen.

- Hierarchical forwarding
    - Assumes a rooted overlay tree
    - Both messages and subscriptions are forwarded by brokers towards the root of the tree
    - Messages flow "downwards" only if a matching subscription had been received along that route

  Routing tables are smaller than subscriptions tables and larger than messages tables.

Cyclic graph (network)

- DHT – Distributed Hash Tables
    - It organizes nodes in a structured overlay allowing efficient routing toward the node having the smaller ID greater or equal than any given ID (the successor)
    - To subscribe for messages having a given subject S
        - Calculate a hash of the subject Hs
        - Use the DHT to route toward the node succ(Hs)
        - While flowing toward succ(Hs) leave routing information to return messages back
    - To publish messages having a given subject S
        - Calculate a hash of the subject Hs
        - Use the DHT to route toward the node succ(Hs)
        - While flowing toward succ(Hs) follow back routes toward subscribers
- Content-based routing - Useful to differentiate between *forwarding* and *routing*
    - Different forwarding strategies:
        - **Per source forwarding (PSF)**
            - ❖ Every source defines a shortest path tree (SPT)
            - ❖ Forwarding table keeps information organized per source
                - → For each source *v* the children in the SPT associated with *v*
                - → For each children *u* a predicate which joins the predicates of all the nodes reachable from *u* along the SPT
        - **Improved per source forwarding (iPSF)**
            - ❖ Same as PSF but leveraging the concept of *indistinguishable sources*
            - ❖ Two sources A e B with SPT T(A) and T(B) are indistinguishable from a node *n* if *n* has the same children for T(A) and T(B) and reaches the same nodes along those children
            - ❖ Some advantages
                - → Smaller forwarding tables
                - → Easier to build them
        - **Per receiver forwarding (PRF)**
            - ❖ The source of a message calculates the set of receivers and add them to the header of the message

            - At each hop the set of recipients is partitioned
            - Two different tables:
                - → The unicast routing table (the one with the next hop)
                - → A forwarding table with the predicate for each node in the network
    - Different strategies to build paths:
        - **Distance Vector (DV)**
            - Builds minimum latency SPTs
            - Use request packets (*config*) and reply to ones (*config response*).
            - Every node acquires a local view of the network (its neighbours in the SPT)
        - **Link-State (LS)**
            - Allows to build SPTs based on different metrics
            - Use packets carrying information about the known state of the network (*Link-State Packet - LSP*) forwarded when a node acquires new information
            - Every node discovers the topology of the whole network
            - SPTs are calculated locally and independently by every node

Complex event processing: CEP systems add the ability to deploy rules that describe how composite events can be generated from primitive (or composite) ones.
There are some open issues with this, for example, find a balance between expressiveness and processing complexity, the process engine, and the distribution, clustered or networked solutions.

3.5 Stream-oriented communication

We have a data stream; information is often organized as a sequence of data units. Time usually does not impact the *correctness* of the communication, just its performance.

Transmission modes

- Asynchronous: the data items in a stream are transmitted one after the other without any further timing constraints (apart ordering)
- Synchronous: there is a max end-to-end delay for each unit in the data stream
- Isochronous: there is max and a min end-to-end delay (bounded jitter)

Stream types:

- Simple
- Complex (composed of substreams)

QoS – Quality of Service represents non-functional requirements like:

- Required bit rate
- Maximum delay to setup the session
- Maximum end-to-end delay
- Maximum variance in delay (jitter)

The DiffServ architecture

- IP is the best effort protocol.
- It offers a Differentiated Services field (aka Type of Service - TOS) into its header
    - 6 bits for the Differentiated Services Code Point (DSCP) field
    - 2 bits for the Explicit Congestion Notification (ECP) field
    - The former encodes the Per Hop Behaviour (PHB)
        - Default
        - Expedited forwarding. Usually less than 30% of traffic and often much less
        - Assured forwarding (further divided into 4 classes)
- Not necessarily supported by Internet routers

One method to enforce **QoS** at the application layer consists in *buffering data*. Even tough, if there is too much congestion, the buffer will become empty, and the application realizes that there was a problem in the network. Thanks to *data interleaving*, if we lose a packet, instead of losing a portion of the video, we are losing frames of the video that are spread in the range of 3-4 seconds. The lost frames can be even reconstructed given the other one that are associated to it.

Synchronization of two or more stream is not so easy.
It could mean different things like left and right channels at CD quality (23µsec of max jitter) or video and audio for lip synch (at 30 fps 33msec of max jitter).
Synchronization may take place at the sender, or the receiver side and it may happen at different layers (application vs. middleware).

4. NAMING

4.1 Basics

Names are used to refer to *entities*. Entities are usually accessed through an *access point* (an address is just a special case of a name).
The same entity can be accessed through several access points at the same time, and it can change its access points during its lifetime. So, it is not convenient to use the address of its access point as a name for an entity, better using *location-independent* names.

- Global names: denotes the same entity, no matter where the name is used.
- Local names: depends on where it is being used.

- Human friendly names
- Machine-friendly names

Identifiers: these are names such that

- they never change (during the lifetime of the entity),
- each entity has exactly one identifier,
- an identifier for an entity is never assigned to another entity.

Using identifiers enables to split the problem of mapping a name to an entity and the problem of locating the entity.

Name resolution: is the process of obtaining the address of a valid access point of an entity having its name.
The way name resolution is performed depends on the nature of the naming schema employed:

- Flat naming
- Structured naming
- Attribute-based naming

4.2 Flat naming

They are simple strings with neither structure nor content.

The name resolution process:

- **Simple solutions**: designed for small environments
  - Broadcast – like ARP where we send a "*find*" message on broadcast channel and only interested host replies.
    Drawback: all hosts must process all "*find*" messages (more traffic and computation)
  - Multicast - same as broadcast, but send to a multicast address to reduce the scope of the search
  - Forwarding pointers (for mobile nodes)
    - Leave reference to the next location at the previous location
    - Chains can become very long

            - Broken links cause chain to break
            - Increased network latency at dereferencing
        - Chain reduction (clean-up) mechanisms can be complex
- **Home-based approaches**: used in Mobile IP and 2.5G cell phone network.
  Rely on one home node that knows the location of the mobile unit. Home is assumed to be stable and can be replicated for robustness. Original IP of the host is effectively used as an identifier.
  Drawback:
    - The extra step towards the home increases latency
    - The home address has to be supported as long as the entity lives
    - The home address is fixed, which means an unnecessary burden when the entity permanently moves to another location
    - Poor geographical scalability (the entity may be next to the client)
- **DHT – Distributed Hash Tables**:
    - *put(id, item)*
    - *item = get(id)*

  Nodes are organized in a structured overlay network (ring, tree, hypercube, etc.)
  Some examples:
    - Chord
        - Nodes and keys are organized in a *logical ring*
        - Each node is assigned a unique *m-bit* identifier (the hash of the IP address)
        - Every item is assigned a unique *m-bit* key (the hash of the item)
        - The item with key *k* is managed (e.g., stored) by the node with the smallest $id \geq k$ (the *successor*)
    - Chord basic lookup
        - Each node keeps track of its successor
        - Search is performed linearly, so if we want to search "*key 80*" and we have nodes with key: 10, 32, 60, 90, when we reach key 90 we stop because we know that key 90 has key 80.
    - Chord "*finger table*"
        - Each node maintains a finger table with *m* entries
        - Entry *i* in the finger table of node *n* is the first node whose id is higher or equal than $n + 2^i$ $(i = 0 \ \dots \ m - 1)$
        - In other words, the i^{th} finger points $\frac{1}{2^{m-1}}$ way around the ring
        - Upon receiving a query for an item with key *k*, a node:
            - Checks whether it stores the item locally
            - If not, forwards the query to the largest node in the 2^{nd} column of its successor table that does not exceed *k*

          (if we have 8 nodes, we use the *mod 8* operation to calculate the successor)
        - Routing table size with $N = 2^m$ nodes? $\log N$ fingers
        - Routing time? Each hop expects to $\frac{1}{2}$ the distance to the desired id
          $\rightarrow O(\log N)$ hops

- Hierarchical approaches
    - The network is divided into domains
    - The root domain spans the entire network, while leaf domains are typically a LAN or a (mobile phone) cell
    - Each domain has an associated directory node that keeps track of the entities in that domain

How to distribute the information?

- The root has entries for every entity
- Entries point to the next sub-domain
- A leaf domain contains the address of an entity (in that domain)
- Entities may have multiple addresses in different leaf domains (replication)

How the lookup works?

- Lookup may start anywhere, where client resides
- If not found, forward lookup to parent
- If entry found, forward lookup to child until a leaf holding the concrete entry is found
- Locality of queries is achieved (good)
- Root has information on all entries (bad)

How updates work?

- Updates start with insert request from new location
- Deleting proceeds from old node up, stopping when a node with multiple children's nodes is reached

Optimizations

- Caching – it is possible to shortcut search if information about mobility patterns is available
- Scalability – root node expected to hold data for all entities. Records are small, but lookups through root create bottleneck.
  The root can be distributed, but then allocation of entities to roots becomes tricky (e.g., if user is in the US, assigning it to the portion of the root maintained in Italy is inefficient)

4.3 Structured naming

In a structured naming system names are organized in a *name space*. A name space is a labelled graph composed of *leaf nodes* and *directory nodes*.

- Leaf node represents a named entity (they include, at least, its identifier/address).
- Directory node has several labelled outgoing edges, each pointing to a different node. A node is a special case of an entity. It has an identifier/address.

Resources are referred through *path names*. It can be:

- absolute

- relative

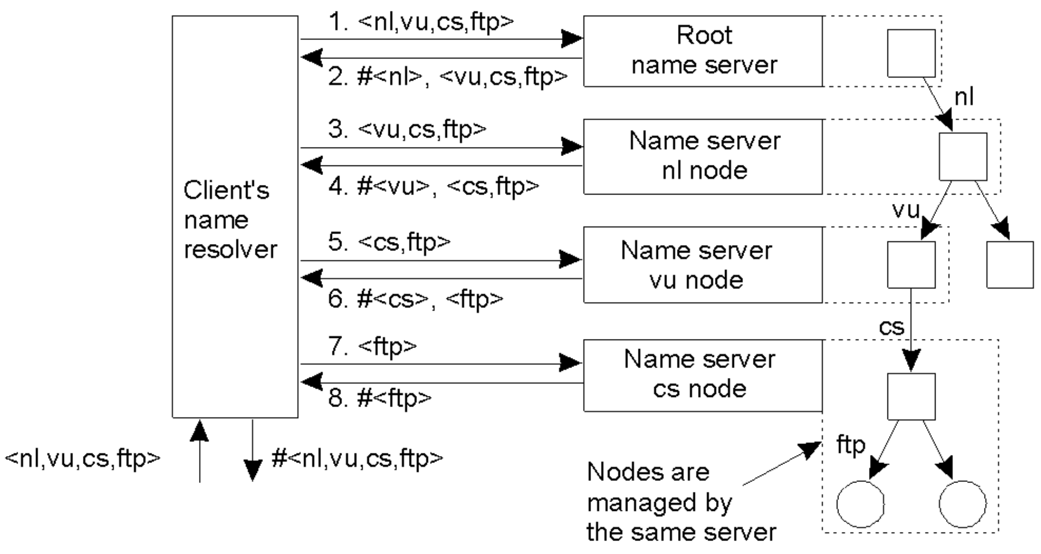
Multiple path names may refer to the same entity (*hard linking*) or leaf nodes may store absolute path names of the entity they refer to instead of their identifier/address (*symbolic linking*).

Name spaces for a large scale, possibly worldwide, distributed system is often distributed among different *name servers*, usually organized hierarchically.

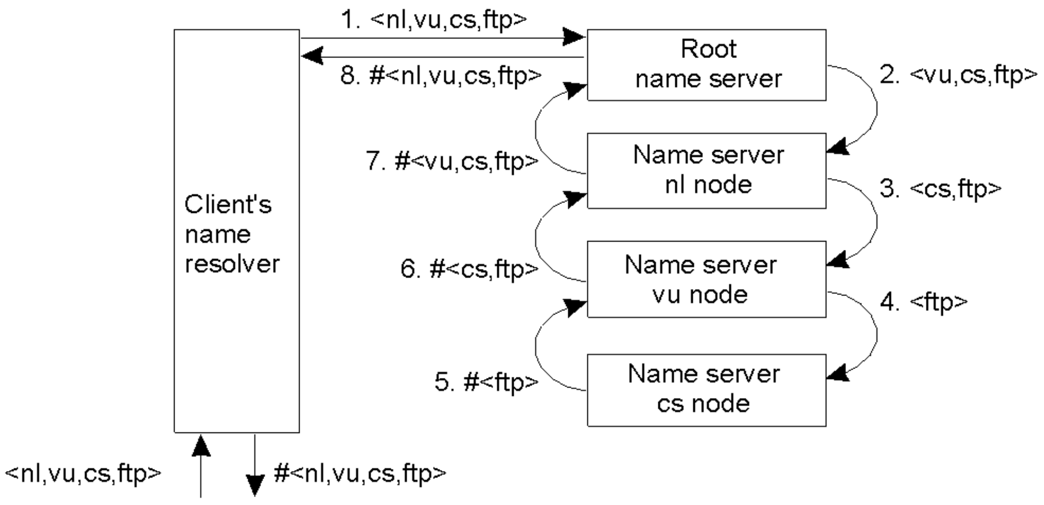
The name space is partitioned into layers:

- **Global level**: high-level directory nodes. Main aspect is that these directory nodes have to be jointly managed by different administrations
- **Administration level**: mid-level directory nodes that can be grouped in such a way that each group can be assigned to a separate administration
- **Managerial level**: low-level directory nodes within a single administration. Main issue is effectively mapping directory nodes to local name servers

Iterative name resolution



Recursive name resolution



Pros of recursive vs. iterative resolution

- Communication costs may be reduced
- Caching is more effective because it can be performed along the whole resolution chain, leading to faster lookups

Cons: higher demand (suspended threads) on each name server

DNS in practice

Name space hierarchically organized as a rooted tree (no hard links).

- Each subtree is named "*domain*", and belongs to a separate authority
- Each name server is responsible for a "*zone*"

Name is case-insensitive, up to 255 characters, max 63 characters per "*label*".

Each node may represent several entities through a set of resource records:

Type of record	**Associated entity**	**Description**
SOA	Zone	Holds information on the represented zone
A	Host	Contains an IP address of the host this node represents
MX	Domain	Refers (symbolic link) to a mail server to handle mail addressed to this node
SRV	Domain	Refers (symbolic link) to a server handling a specific service, e.g., http
NS	Zone	Refers (symbolic link) to a name server that implements the represented zone
CNAME	Node	Symbolic link with the primary name of the represented node
PTR	Host	Contains the canonical name of a host, for reverse lookups
HINFO	Host	Holds information on the host this node represents
TXT	Any kind	Contains any entity-specific information considered useful

Clients can request the resolution mode, but servers are not obliged to comply (global name servers typically support only iterative resolution).
Global servers are mirrored, and IP anycast is used to route queries among them.

Caching and replication are massively used:

- secondary DNS servers are periodically (e.g., twice per day) brought up to date by the primary ones
- A TTL attribute is associated to information, determining its persistence in the cache

Transient (days at the global level) inconsistencies are allowed. (Only host information is stored, but in principle other information could be stored)

What if a host is allowed to "*move*"?

- Not a problem if it stays within the original domain; just update the database of the domain name servers.
- If a host (e.g., ftp.elet.polimi.it) moves to an entirely different domain (e.g., cs.wustl.edu)
    - Better keep its name because applications and users are relying on it
    - DNS servers of elet.polimi.it could provide the IP address of the new location (lookups not affected further updates no longer "localized"; managerial layer no longer efficient)
    - DNS servers of elet.polimi.it could provide the name of the new location (essentially a *symbolic link*). Like the forwarding pointers approach.

4.4 Attribute based naming

Refer to entities not with their name but with a set of attributes, which code their properties.

Each entity has a set of associated attributes.
The name system can be queried by searching for entities given the values of (some) of their attributes.

Attribute based naming systems are usually called *directory services*.

A common approach to implement distributed directory services is to combine structured with attribute-based naming:

- LDAP (Lightweight Directory Access Protocol)
  LDAP directory consists of several records (directory entries)
    - Each is made as a collection of <attribute, value> pairs
    - Each attribute has a type
    - Both single-valued and multiple-valued attributes exist

  The collection of all records in a LDAP directory service is called *Directory Information Base* – DIB. Each record has a unique name defined as a sequence of *naming attributes*.

  This leads to build a *directory information tree*.

  When dealing with large scale directories, the DIB is usually partitioned according to the DIT structure (as in DNS)

    - Each server is known as a *Directory Service Agent* – DSA
    - The client is known as *Directory User Agent* – DUA

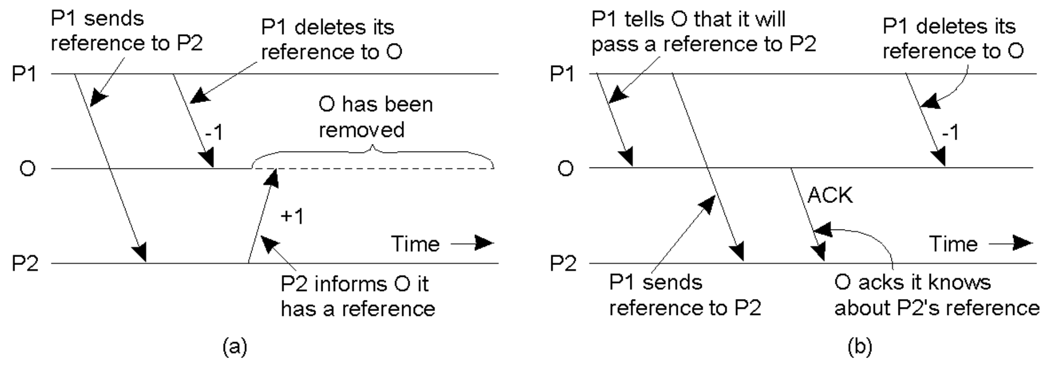
  What LDAP adds to a standard hierarchical naming schema is its searching ability.

4.5 Removing unreferenced entities

Entities accessed through state bindings should be removed. Automatic garbage collection is common in conventional systems.
Distribution greatly complicates matters, due to lack of global knowledge about who's using what, and to network failures.

- **Reference counting**
    - The object keeps track of how many other objects have been given references

- Race condition when passing references among processes (not present in non-distributed systems):


P1 sends reference to P2
P1 deletes its reference to O
P1
O
P2
-1
O has been removed
+1
Time
P2 informs O it has a reference
(a)
P1 tells O that it will pass a reference to P2
P1 deletes its reference to O
ACK
-1
Time
P1 sends reference to P2
O acks it knows about P2's reference
(b)

- Passing a reference requires now three messages (potential performance issue in large-scale systems).

- **Weighted reference counting**
  - Tries to circumvent the race condition by communicating only counter decrements
  - Requires an additional counter
  - Removing a reference subtracts the proxy partial counter from the total counter of the skeleton: *when the total and partial weights become equal*, the object can be removed
  - Problem: only a fixed number of references can be created

  Using indirection removes the limitation on the overall number of references but introduces an additional hop to access the target object.
- **Reference listing**: instead of keeping track of the number of references, keep track of the identities of the proxies.
  - Advantage 1: insertion/deletion of a proxy is idempotent
    - Insertion and deletion of references must still be acknowledged, but requests can be issued multiple times with the same effect
    - Non-reliable communication can be used
  - Advantage 2: easier to maintain the list consistent with regards to network failures
    - E.g., by periodically “pinging” clients (potential scalability problem)
  - Still suffers from race conditions when copying references

  Used by Java RMI.

How to detect entities disconnected from the root set?
Tracing-based garbage collection techniques: require knowledge about all entities, therefore they have inherent poor scalability.

So, we use ***Mark-and-sweep*** on uniprocessor systems:

- First phase marks accessible entities by following references
  - Initially all nodes *white*
  - A node is coloured *grey* when reachable from a root (but some of its references still need to be evaluated)
  - A node is coloured *black* after it turned grey, and all its outgoing references have been marked grey

- Second phase exhaustively examines memory to locate entities not marked and removes them
    - Garbage collects all white nodes

In a distributed system we have a ***distributed mark-and-sweep***.

- Garbage collectors run on each site. (Initially all processes are marked white)
- Marking process
    - An object in process *P*, and reachable from a root in *P*, is marked grey together with all the proxies in it
    - When a proxy *q* is marked grey, a message is sent to its associated skeleton (and object)
        - An object is marked grey as soon as its skeleton is
        - Recursively, all proxies in the object are marked grey
    - An acknowledgment is expected before turning *q* black
- Sweep process:
    - All white objects are collected locally
    - Can start as soon as all local objects are either black or white
- Requires reachability graph to remain stable!
    - Distributed transaction blocking the entire system

5. SYNCHRONIZATION

5.1 Synchronization in distributed systems: An Introduction

The problem of synchronizing concurrent activities arises also in non-distributed systems. However, distribution complicates matters:

- Absence of a global physical clock
- Absence of globally shared memory
- Partial failures

Time plays a fundamental role in many applications.

5.2 Synchronization physical clocks

Computer clocks are not clocks; they are timers.

To guarantee synchronization:

- Maximum *clock drift rate* ρ is a constant of the timer
- Maximum allowed *clock skew* δ is an engineering parameter
- If two clocks are drifting in opposite directions, during a time interval Δt they accumulate a skew of $2\rho\Delta t$
    - resynch needed at least every $\delta / 2\rho$ seconds

The problem is either:

- Synchronize all clocks against one, usually the one with external, accurate time information (accuracy)
- Synchronize all clocks among themselves (agreement)

GPS

How GPS works:

- Position is determined by triangulation from a set of satellites whose position is known
- Distance can be measured by the delay of signal
- But satellite and receiver clock must be in sync
- Since they are not, we must take clock skew into account

But we have some problems like atomic clocks in the satellites are not perfectly in sync, Earth is not spherical, the position of satellites is not known precisely, the receiver's clock has a finite accuracy, the signal propagation speed is not constant, etc.

Cristian's algorithm (1989)

- Periodically, each client sends a request to the time server
- Messages are assumed to travel fast

We have some problems:

- Time might run backwards on client machine. Therefore, introduce change gradually (e.g., advance clock 9ms instead of 10ms on each clock tick)

- It takes a non-zero amount of time to get the message to the server and back
    - Measure round-trip time and adjust, e.g., $T_1 = C_{UTC} + \frac{T_{round}}{2}$
    - Average over several measurements

Berkeley algorithm

It collects the time from all clients, averages it, and then retransmits the required adjustment.

Network Time Protocol (NTP)

Designed for UTC synch over large-scale networks, used in practice over the Internet, on top of UDP.

Hierarchical synchronization subnet organized in strata

- Servers in stratum 1 are directly connected to a UTC source
- Lower strata (higher levels) provide more accurate information
- Leaf servers execute in users' workstations
- Connections and strata membership change over time

Synchronization mechanisms

- Multicast (over LAN)
    - Servers periodically multicast their time to other computers on the same LAN
- Procedure-call mode (similar to Cristian's algorithm)
- Symmetric mode
    - For higher levels that need the highest accuracies

5.3 Logical time

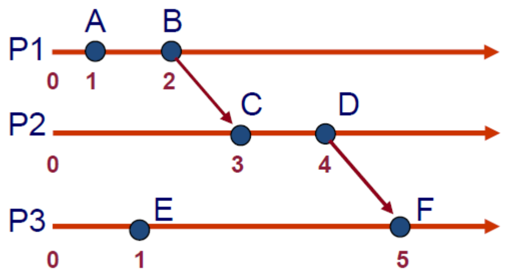
In many applications it is sufficient to agree on a time, even if it is not accurate. What matters is often the ordering and causality relationships of events, rather than the timestamp itself.

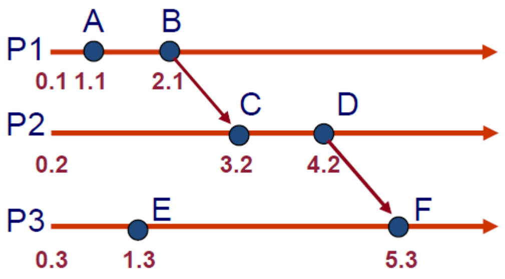
Scalar clocks

- Let define the *happens-before* relationship $e \rightarrow e'$, as follows:
- If events e and e' occur in the same process and e occurs before e', then $e \rightarrow e'$
- If $e = send(msg)$ and $e' = recv(msg)$, then $e \rightarrow e'$ → is transitive

- If neither $e \rightarrow e'$ nor $e \rightarrow e'$, they are concurrent ($e||e'$)
- The happens-before relationship captures *potential causal ordering* among events
    - Two events can be related by the happens-before relationship even if there is no real (causal) connection among them
    - Also, since information can flow in ways other than message passing, two events may be causally related even neither of them happens-before the other
- Lamport invented a simple mechanism by which the happened before ordering can be captured numerically
- Each process *p* keeps a logical scalar clock L_i
    - L_i starts at zero
    - L_i is incremented before p_i sends a message
    - Each message sent by p_i is timestamped with *Li*
    - Upon receipt of a message, p_i sets L_i to: $MAX(masg\ timestamp, L_i) + 1$

- It can easily be shown, by induction on the length of any sequence of events relating two events *e* and *e'*, that: $e \rightarrow e' \Rightarrow L(e) < L(e')$
- Note that only partial ordering is achieved. Total ordering can be obtained trivially by attaching process IDs to clocks





- **Totally ordered multicast using scalar clocks**
  Updates are important especially the order in which the operations are done.
  Imagine having a deposit of 100$ and the bank adds 1% interest. We have 2 DBs and we want to add 1000$.
  So, we could have
  - $(100 + 1000) + 1\% = 1111\$$ (right order)
  - $100 + (1000 + 1\%) = 1110\$$ (wrong order)

  So, the order in which the operations are done is very important.
  Using scalar clocks (assuming reliable and FIFO links):
  - Messages are sent and acknowledged using multicast
  - All messages (including acks) carry a timestamp with the sender's scalar clock
  - Scalar clocks ensures that the timestamps reflect a consistent global ordering of events
  - Receivers (including the sender) store all messages in a queue, ordered according to its timestamp
  - Eventually, all processes have the same messages in the queue
  - A message is delivered to the application only when it is at the highest in the queue and all its acks have been received
  - Since each process has the same copy of the queue, all messages are delivered in the same order everywhere

Vector clocks

In vector clocks each process p_i maintains a vector V_i of N values (N = #processes) such that:

- $V_i[i]$ is the number of events that have occurred at P_i
- If $V_i[j] = k$ then P_i knows that *k* events have occurred at P_j

Rules for updating the vectors:

- Initially, $V_i[j] = 0 \; \forall \; i, j$
- Local event at P_i causes an increment of $V_i[i]$
- P_i attaches a timestamp $t = V_i$ in all messages it sends (incrementing $V_i[i]$ just before sending the message, according to previous rule)

- When *P*$_i$ receives a message containing t, it sets $V_i[j] = \max(V_i[j], t[j])\ \forall\, j \neq i$ and then increments *V*$_i$[i]

Definitions (partial ordering):

- $V = V' \leftrightarrow V[j] = V'[j], \forall\, j$
- $V \leq V' \leftrightarrow V[j] \leq V'[j], \forall\, j$
- $V < V' \leftrightarrow V \leq V' {}^\wedge V \neq V'$
- $V' \leftrightarrow \neg(V < V')^\wedge\neg(V' < V)$

Determining causality:

- $e \rightarrow e' \leftrightarrow V(e) < V(e')$
- $e \mid\mid e' \leftrightarrow V(e) \mid\mid V(e')$

What about *casual delivery*?
A slight variation of vector clocks can be used to implement causal delivery of messages in a totally distributed way.

Messages and replies sent (using reliable and FIFO ordered channels) to all the boards in parallel. We need to preserve the ordering only between messages and replies.

- Totally ordered multicast is too strong
  - If M1 arrives before M2, it does not necessarily mean that the two are related
- Using vector clocks:
  - Variation: increment clock only when sending a message. On receive, just merge, not increment
  - Hold a reply until the previous messages are received:
    - $ts(r)[j] = V_k[j] + 1$
    - $ts(r)[i] \leq V_k[i]\ \forall\, i \neq j$

5.4 Mutual exclusion

Required to prevent interference and ensure consistency of resource access.

Requirements:

- Safety property (at most one process may execute in the critical section at a time)
- Liveness property (all requests to enter/exit the critical section eventually succeed, no deadlock, no starvation)
- Optional (if one request happened-before another, then entry is granted in that order)

Assumptions:

- Reliable channels and processes

With scalar clocks:

- To request access to a resource
  - A process P*i* multicasts a resource request message *m*, with timestamp T*m*, to all processes (including itself)
  - Upon receipt of *m*, a process P*j*:

        - If it does not hold the resource and it is not interested in holding the resource, P*j* sends an acknowledgment to P*i*
        - If it holds the resource, P*j* puts the requests into a local queue ordered according to T*m* (process ids are used to break ties)
        - If it is also interested in holding the resource and has already sent out a request, P*j* compares the timestamp Tm with the timestamp of its own requests
            - If the T*m* is the lowest one, P*j* sends an acknowledgement to P*i*, otherwise it put the request into the local queue above
- On releasing the resource, a process P*i* acknowledges all the requests queued while using the resource
- A resource is granted to P*i* when its request has been acknowledged by all the other processes

With token ring solution:

- Processes are logically arranged in a ring, regardless of their physical connectivity
- Access is granted by a token that is forwarded along a given direction on the ring
    - A process not interested in accessing the resource forwards the token
    - Resource access is achieved by retaining the token
    - Resource release is achieved by forwarding the token

A centralized architecture is the best.

5.5 Leader election

Many distributed algorithms require a process to act as a coordinator (or some other special role) like for server-based mutual exclusion.

Minimal assumptions:

- Nodes are distinguishable
- Processes know each other and their IDs

Typically use the identifier (the process with the highest ID becomes the leader) or some other measure to elect the winner or the non-crashed process with the highest ID.

Bully election algorithm

Assumptions: reliable links and it is possible to decide who has crashed (synchronous system).

- When any process *P* notices that the actual coordinator is no longer responding to the requests, it initiates an election
- *P* sends an *ELECT* message, including its ID, to all other processes with higher IDs
- If no-one responds, *P* wins and sends a *COORD* message to the processes with lower IDs
- If a process *P'* receives an *ELECT* message it responds (stopping the former candidate) and starts a new election (if it has not started one already)
- If a process that was previously down comes back up, it holds an election
    - If it happens to be the highest-numbered process currently running it wins the election and takes over the coordinator's job (hence the name of the algorithm)

Ring-based algorithm

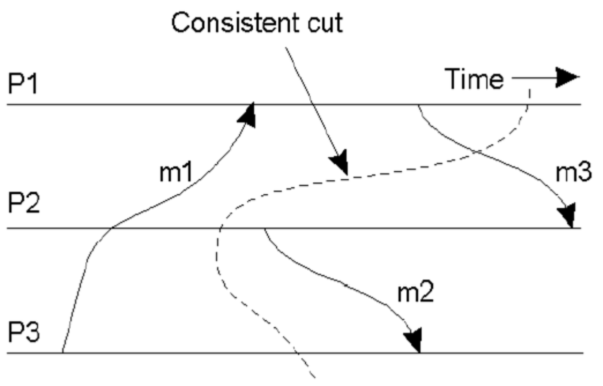
- When a process detects a leader failure, it sends an *ELECT* message containing its ID to the closest alive neighbour
- Upon receipt an *ELECT* message a process *P*:
    - If *P* is not in the message, add *P* and propagate to next alive neighbour
    - If *P* is in the list, change message type to *COORD*, and re-circulate
- On receiving a *COORD* message, a node considers the process with the highest ID as the new leader (and is also informed about the remaining members of the ring)
- Multiple messages may circulate at the same time and eventually converge to the same content

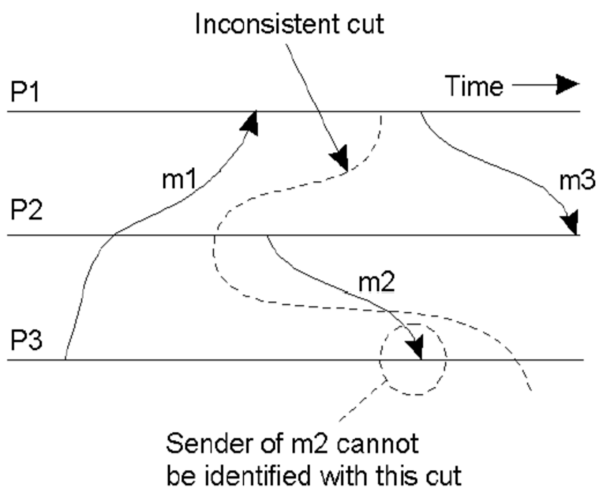
5.6 Collecting global state

The global state of a distributed system consists of the local state of each process (depends on the application), together with the message in transit over the links. Useful to know for distributed debugging, termination detection, deadlock detection, etc.

It would be easy if we could access a global clock, but we do not have one and we must accept recording the state of each process at, potentially, different times.

A *distributed snapshot* reflects a (consistent, global) state in which the distributed system might have been. Care must be taken when reconstructing the global state to preserve consistency. We use a conceptual tool: *Cut*.





Formally a *cut* of a system *S* composed of *N* processes *p1, ...,* pn can be defined as the union of the histories of all its processes up to a certain event $C = h_1^{k1} \cup h_2^{k2} \cup \dots \cup h_n^{kn}$ where
$h_i^{ki} = \langle e_i^0, e_i^1 \dots e_i^{ki} \rangle$

A cut *C* is consistent if for any event *e* it includes, it also includes all the events that happened before *e*.

Distributed snapshot

Assume FIFO, reliable links/nodes + strongly connected graph.
The distributed snapshot algorithm does not require blocking of the computation.

- Any process p may initiate a snapshot by

    - Recording its internal state
    - Sending a token on all outgoing channels (this signals a snapshot is being run)
    - Start recording a local snapshot
- Upon receiving a token, a process *q*
    - If not already recording local snapshot
        - Records its internal state
        - Sends a token on all outgoing channels
        - Start recording a local snapshot (see above)
    - In any case stop recording incoming message on the channel the token arrived along
- Recording messages
    - If a message arrives on a channel which is recording messages, record the arrival of the message, then process the message as normal
    - Otherwise, just process the message as normal
- Each process considers the snapshot ended when tokens have arrived on all its incoming channels
    - Afterwards, the collected data can be sent to a single collector of the global state

What happens if the snapshot is started at more than one location at the same time?
Easily dealt with by associating an identifier to each snapshot, set by the initiator.

There are some variations like incremental snapshots.

Termination detection

We want to know when a computation has completed or deadlocked (no more useful work can be done).

- All processes should be idle
- There should be no messages in the system
    - Messages need to be processed, i.e., some process must become non-idle

Can a distributed snapshot be used? Yes, but channels must be empty when finished.

In a diffusing computation, initially, all processes are idle except the *init* process.
A process is activated only by a message being sent to it and the termination condition is when processing is complete at each node, and there are no more messages in the system.

Dijkstra – Scholten (for diffusing computations)

- Create a tree out of the active processes
- When a node finishes its processing and it is a leaf, it can be pruned from the tree
- When only the root remains, and it has completed processing, the system has terminated

How to create a tree?

- Each node keeps track of nodes it sends a message to (those it may have woken up), its children
- If a node was already awake when the message arrived, then it is already part of the tree, and should not be added as a child of the sender

- When a node has no more children and it is idle, it tells its parent to remove it as a child

COMPARISON	
Distributed snapshot	**Dijkstra - Scholten**
• Overhead is one message per link • Plus, cost to collect result • If system not terminated, need to run again!	• Overhead depends on the number of messages in the system ○ Acknowledgments sent when already part of network and when become idle ○ Can be added to network more than once, so this value is not fixed • Does not involve never-activated processes • Termination detected when last ack received

5.7 Distributed transactions

Protect a shared resource against simultaneous access by several concurrent processes. *Transactions* are sequences of operations, defined with appropriate programming primitives like `BEGIN_TRANSACTION`, `END_TRANSACTION`, `READ`, `WRITE`, etc.

All transactions must satisfy the ACID properties:

- Atomic, to the outside world, the transaction happens indivisibly.
- Consistent, the transaction does not violate system invariants.
- Isolated, concurrent transactions do not interfere with each other.
- Durable, once a transaction commits, the changes are permanent.

There are 3 types of transaction:

1. Flat
2. Nested, a transaction that is composed by two sub-transactions done on two different databases.
   They operate on a private copy of the data. If it aborts the private copy disappears, if it commits, the modified private copy is available to the next sub-transaction.
3. Distributed, essentially two sub-transactions on a distributed database.

Atomicity

- Private workspace
  - Copy what the transaction modifies into a separate memory space, creating *shadow blocks* of the original file.
    If the transaction is aborted, this private workspace is deleted, otherwise they are

copied into the parent's workspace.
(Works fine also for the local part of distributed transactions)

- Writehead log
  - Files are modified in place (commit is fast), but a log is kept with
    - Transaction that made the change
    - Which file/block
    - Old and new values
  - After the log is written successfully, the file is modified.
  - If transaction succeeds, commit written to log; if it aborts, original state restored based on logs, starting at the end (rollback).

Concurrency

To achieve atomicity, we need a transaction manager, instead, to determine which transaction can pass an operation to the data manager and when, we need a scheduler.
It can be distributed if we have distributed transactions, and it leads us to *serializable* transactions.

Transaction systems must ensure operations are interleaved correctly, but also free the programmer from the burden of programming mutual exclusion.

Two ways to *serialize* transactions:

- **Two-Phase Locking (2PL)**
  - When a process needs to access data, it requests the scheduler to grant a lock. The scheduler tests whether the requested operation conflicts with another that has already received the lock: if so, the operation is delayed. Once a lock for a transaction T has been released, T can no longer acquire it.
  - 2PL Strict – releasing the locks all at the same time, prevents cascaded aborts by requiring the shrink phase to take place only after transaction termination.
  - Centralized 2PL – centralized lock manager
  - Primary 2PL – multiple lock managers. Each data item has a primary copy on a host, the lock manager on that host is responsible for granting locks and the transaction manager is responsible for interacting with the data managers.
  - Distributed 2PL – data may be replicated on multiple hosts. The lock manager on a host is responsible for granting locks on the local replica and for contacting the (local) data manager.
- **Pessimistic timestamp ordering**
  - Assign a timestamp to each transaction. We have to write and read operation on data *x* with a timestamp; $ts_{wr}(x_i)$
  - The schedule operates as follow:
    - When receives *write(T,x)* at *time=ts*
      - ❖ If $ts > ts_{rd}(x)$ and $ts > ts_{wr}(x)$ perform tentative write xi with timestamp $ts_{wr}(xi)$
      - ❖ else abort *T* since the write request arrived too late
    - Scheduler receives *read(T,x)* at *time=ts*
      - ❖ If $ts > ts_{wr}(x)$

        - → Let x_{sel} be the latest version of x with the write timestamp lower than *ts*
        - → If x_{sel} is committed perform read on x_{sel} and set $ts_{rd}(x) = max(ts, ts_{rd}(x))$
        - → else wait until the transaction that wrote version x_{sel} commits or abort then reapply the rule
      - ❖ else abort *T* since the read request arrived too late
  - Aborted transactions will reapply for a new timestamp and simply retry.
  - Deadlock-free.
- **Optimistic timestamp ordering**
  - Based on the assumption that conflicts are rare, therefore do what you want without caring about others, fix conflicts later.
    - Stamp data items with start time of transaction
    - At commit, if any items have been changed since start, transaction is aborted, otherwise committed
  - Best implemented with private workspaces
  - Deadlock-free, allows maximum parallelism
  - Under heavy load, there may be too many rollbacks
  - Not widely used, especially in distributed systems

Distributed deadlocks

Same concept as in conventional systems but worse to deal with, since in a distributed system resources are spread out.

4 approaches:

- Ignore the problem
- Detection and recovery, typically by killing one of the processes
- Prevention
- Avoidance (never in distributed systems)

- Centralized deadlock detection
  - Each machine maintains a resource graph for its own resources and reports it to a coordinator.
  - Options to collect information
    - Whenever an arc is added or deleted, a message is sent to the coordinator with the update
    - Periodically, every process sends a list of arcs added or deleted since the last update
    - Coordinator can request information on-demand
  - None works well, because of false deadlocks
- Distributed deadlock detection
  - There is no coordinator in charge of building the global wait-for graph
  - Processes are allowed to request multiple resources simultaneously

    - When a process gets blocked, it sends a probe message to the processes holding resources it wants to acquire
        - Probe message: (initiator, sender, receiver)
        - A cycle is detected if the probe makes it back to the initiator
    - How to recover when a deadlock is detected?
        - Initiator commits suicide but many processes may be unnecessarily aborted if more than one initiator detects the loop
        - The initiator picks the process with the higher identifier and kills it but requires each process to add its identifier to the probe
- Distributed prevention (never used in distributed systems)
    - Using global timestamps (*wait-die algorithm*)
        - When a process A is about to block for a resource that another process B is using, allow A to wait only if A has a lower timestamp (it is older) than B; otherwise kill the process A
        - Following a chain of waiting processes, the timestamps will always increase (no cycles)
        - In principle, could have the younger wait
    - If a process can be preempted, i.e., its resource taken away, an alternative can be devised (*wound-wait algorithm*)
        - Preempting the young process aborts its transaction, and it may immediately try to reacquire the resource, but will wait
        - In wait-die, young process may die many times before the old one releases the resource

6. FAULT TOLERANCE

6.1 Introduction

- Availability
  - We want the system to be ready for use as soon as we need it
- Reliability
  - The system should be able to run continuously for long time
- Safety
  - It should cause nothing bad to happen
- Maintainability
  - It should be easy to fix

A system is said to be *fault tolerant* if it can provide its services even in the presence of faults.

A system *fails* when it is not able provide its services.
A failure is the result of an *error*.
An error is caused by a *fault*.

Faults can be classified according to the frequency at which they occur:

- Transient faults occur once and disappear
- Intermittent faults appear and vanish with no apparent reason
- Permanent faults continue to exist until the failed components are repaired

Different types of failure:

- Omission failures
  - Processes: fail-safe (sometimes wrong, but easily detectable output), fail-stop (detectable crash), fail-silent (undetectable crash)
  - Channels: send omission, channel omission, receive omission
- Timing failures (apply to synchronous systems, only)
  - Occur when one of the time limits defined for the system is violated
- Byzantine (or arbitrary) failures
  - Processes: may omit intended processing steps or add more
  - Channels: message content may be corrupted, non-existent messages may be delivered, or real messages may be delivered more than once

The key technique to mask a failure is *redundancy*.

6.2 An example: Client / Server communication

Reliable point-to-point communication is usually provided by the TCP protocol, which masks omission failures using acks and retransmissions.
Ideally RPC should provide the semantics of procedure calls in a distributed setting.
A number of problems arise:

- Server cannot be located – the client can handle this as an exception.
- Lost Requests – if too many requests are lost, the client can conclude the server is down

- Server crashes – the problem is that the client cannot tell when the server crashed. Assuming the client knows the server crashed, it can use four strategies
    - Always reissue request
    - Never reissue request
    - Reissue only when message not received
    - Reissue only when message received
- Lost replies – Was the reply lost? Or did the request not arrive? Some requests are idempotent others are not. The server should cache clients' requests so as not to repeat duplicate ones.
- Client crashes – A computation started by a dead client is called an *orphan*.
  The server must get rid of them because they can still consume resources
    - Extermination: RPC's are logged by the client and orphans killed (on client request) after a reboot -> costly (logging each call) and problem with grand orphans and network partitions
    - Reincarnation: when a client reboots it starts a new epoch and sends a broadcast message to servers, who kill old computations started on behalf of that client (including grand orphans). Replies sent by orphans bring an obsolete epoch and can be easily discarded
    - Gentle reincarnation: as before but servers kill old computations only if owner cannot be located
    - Expiration: Remote computation expire after some time. Clients wait to reboot to let remote computations to expire

6.3 Protection against process failures

Redundancy can be used to mask the presence of faulty processes, with redundant process groups.

Two possible organizations:

- Flat group – a ring with all processes connected each other
- Hierarchical group – there is a coordinator and some workers

How large should a group be?

- If processes fail silently, then **k+1 processes** allow the system to be **k-fault tolerant**
- If failures are Byzantine, matters become worse: **2k+1 processes** are required to achieve **k-fault tolerance** (to have a working voting mechanism)

A number of tasks may require that the members of a group agree on some decision before continuing because we are dealing with faults, we want *all nonfaulty processes to reach an agreement*.

More precisely, we have the following conditions:

- Each process starts with some initial value
- All non-faulty processes have to reach a decision based on these initial values
- The following properties must hold

- Agreement: No two processes decide on different values
- Validity: If all processes start with the same value v, then v is the only possible decision value
- Termination: All non-faulty processes eventually decide

The consensus **is not possible** in the presence of arbitrary communication failures.

FloodSet algorithm

- Let v0 be a pre-specified default value
- Each process maintains a variable W (subset of V) initialized with its start value
- The following steps are repeated for f+1 rounds
  - Each process sends W to all other processes
  - It adds the received sets to W
- The decision is made after f+1 rounds
  - if |W| = 1: decide on W's element
  - if |W| > 1: decide on v0 (or use the same function to decide, e.g., max(W))
- Remember: each process may stop in the middle of the send operation

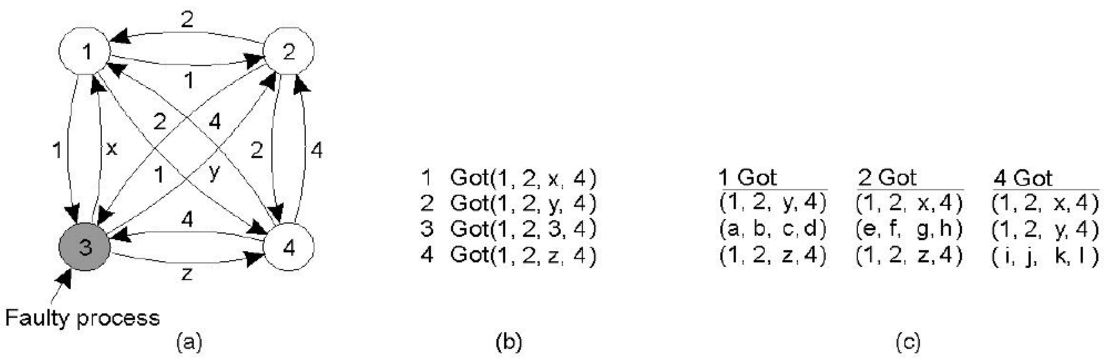
Let us now make things more complex by introducing byzantine failures. (Now processes can not only stop but also exhibit any arbitrary behaviour, sending arbitrary messages, performing arbitrary state transitions and so on)

Our problem is now defined as follows:

- Agreement: No **two non-faulty** processes decide on different values
- Validity: If **all non-faulty** processes start with the same value v, then v is the only possible decision value for all non-faulty processes
- Termination: All non-faulty processes eventually decide

So, imagine having 4 generals and 1 traitor, here there are the steps for Lamport's algorithm:

- Send troop strength to others
- Form a vector with received values
- Send vector to others
- Compute vector using majority for each vector position



```
1  Got(1, 2, x, 4)
2  Got(1, 2, y, 4)
3  Got(1, 2, 3, 4)
4  Got(1, 2, z, 4)
```

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

Lamport (1982) showed that if there are *m* traitors, *2m+1* loyal generals are needed for an agreement to be reached, for a total of *3m+1*.

So far, we have considered synchronous systems, but a real distributed system is inherently asynchronous.
Let us consider the problem of reaching an agreement between *n* processes with 1 faulty process in an asynchronous system.

Fischer, Lynch, and Paterson proved that **no solution exists**: "*Impossibility of Distributed Consensus with one Faulty Process*" (FLP Theorem). The result was proved in the case of *crash failures*, but what happens in case of *byzantine failures*? If a solution existed for byzantine failures that solution would also work for crash failures.

6.4 Reliable group communication

- Groups are fixed, and processes non-faulty
    - All group members should receive the multicast (not necessarily in the same order)
    - Easy to implement on top of unreliable multicast
        - Positive acknowledgements – Large number of ack messages
        - Negative acknowledgments – Fewer messages but sender has to cache messages
    - Basic approach
        - Each receiver sends an ACK to the sender but there could be a problem on the network due to the load of the ACKs (ACK implosion)
    - Scalable reliable multicast
        - Each receiver sends a NACK to all others with a different delay, so everyone has to process NACKs. Then, just one NACK will be received by the sender.
    - Hierarchical feedback control
        - Receivers are organized in groups headed by a coordinator and groups are organized in a tree routed at the sender. The coordinator can adopt any strategy within its group.
          The problem is that the hierarchy (tree) has to be constructed and maintained.
- Case of faulty processes
    - Close synchrony
        - Any two processes that receive the same multicast messages or observe the same group membership changes to see the corresponding events in the same order.
        - A multicast to a process group to be delivered to its full membership. The send and delivery events should be considered to occur as a single, instantaneous event.
        - Unfortunately, close synchrony cannot be achieved in the presence of failures.
    - Virtual synchrony – we need a model which distinguishes between receiving and delivering a message because, even if we can detect failures correctly, we cannot know whether a failed process has received and processed a message.
        - A *group view* is the set of processes to which a message should be delivered as seen by the sender at sending time.

    - The minimal ordering requirement is that a group view change be delivered in a consistent order with respect to other multicasts and with respect to each other.
    - We say that a *view* change occurs when a process joins or leaves the group, possibly crashing.
    - All multicasts must take place between view changes
        - We can see view changes as another form of multicast messages (those announcing changes in the group membership).
        - We must guarantee that messages are always delivered before or after a view change.
        - If the view change is the result of the sender of a message m leaving the message is either delivered to all group members before the view change is announced or it is dropped.
    - Multicasts take place in epochs separated by group membership changes.

Retaining the virtual synchrony property, we can identify different orderings for the multicast messages.

- Unordered multicasts
- FIFO-ordered multicasts
- Casually ordered multicasts

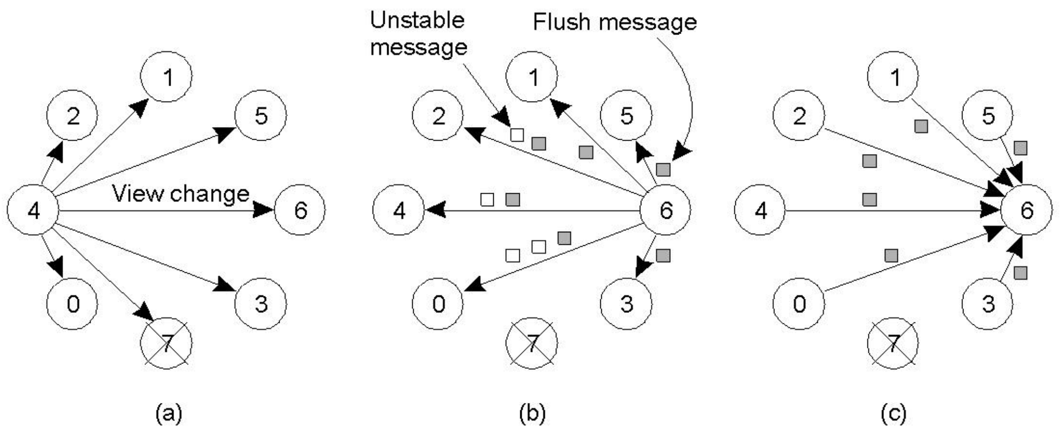
In addition, the above orderings can be combined with a total ordering requirement. Whatever ordering is chosen, messages must be delivered to every group member in the same order.

- *Atomic multicast* is defined as a virtually synchronous reliable multicast offering totally ordered delivery of messages.

Implementation of Virtual Synchrony

- We assume that processes are notified of view changes by some (possibly distributed component).
- When a process receives a view change message, stop sending new messages until the new view is installed, instead it multicasts all pending unstable messages to the non-faulty members of the old view, marks them as stable and multicasts a flush message.
- Eventually all the non-faulty members of the old view will receive the view change and do the same (duplicates are discarded).
- Each process installs the new view as soon as it has received a flush message from each other process in the new view. Now it can restart sending new messages.

If we assume that the group membership does not change during the execution of the protocol above, we have that at the end all non-faulty members of the old view receive the same set of messages before installing the new view. The protocol can be extended to account the case of group changes while in progress.



Process 4 notices that process 7 has crashed, sends a view change.
Process 6 sends out all its unstable messages, followed by a flush message.
Process 6 installs the new view when it has received a flush message from everyone else.

6.5 Recovery techniques

When processes resume working after a failure, they have to be taken back to a correct state. We have two ways to recover it:

- Backward recovery – The system is brought back to a previously saved correct state (checkpointing and logging)
- Forward recovery – The system is brought into a new correct state from which execution can be resumed (error correction codes)

Checkpointing

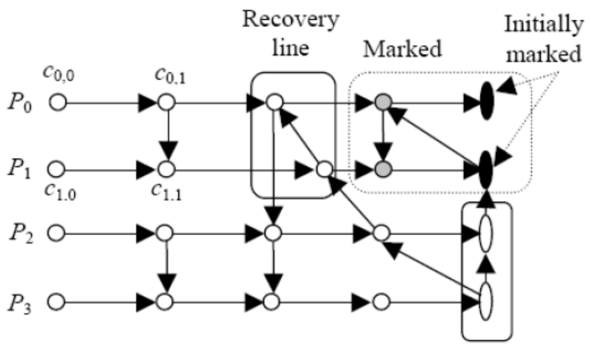
It consists in periodically saving the distributed state of the system to stable storage, so we need to find a consistent cut.

The most trivial solution is to have each process *independently* record its own state. We may need to roll back to previous checkpoint until we find a set of checkpoints (one per process) which result in a consistent cut.
This is not so easy to implement, in fact, if process A rolls back to a checkpoint before interval *i*, all processes must be rolled back to an interval that does not depend on *i*.

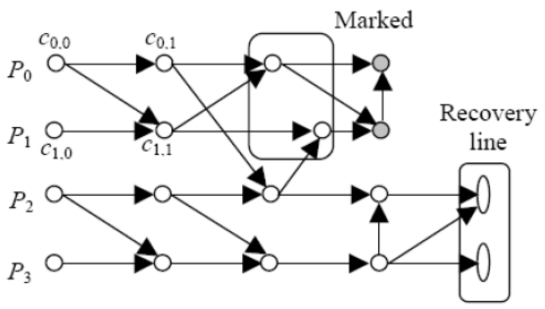
When a failure occurs, the recovering process broadcasts a dependency request to collect dependency information. All processes stop and send their information and the recovering process computes the recovery line and sends the corresponding rollback request.

The recovery line can be computed using two approaches:

- Rollback dependency graph
- Checkpoint graph



Rollback-dependency graph



Checkpoint-dependency graph

In the *rollback-dependency graph* we draw an arrow between two checkpoints $c_{j,x}$ and $c_{j,y}$ if either

- $i = j\ and\ y = x + 1$
- $i \neq j$ and a message is sent from $i_{j,x}\ to\ i_{j,y}$

The recovery line is computed by marking the nodes corresponding to the failure states and then marking all those which are reachable from one of them. These marked nodes are no good. Each process rolls back to the last unmarked checkpoint.

An equivalent approach can be used on the *checkpoint graph*. Take the latest checkpoints that do not have dependencies among them.

The complexity of the previous algorithm suggests that independent checkpointing is not so convenient; the solution is to *coordinate checkpoints*.

A simple solution is the following:

- Coordinator sends CHKP-REQ
- Receivers take checkpoint and queue other outgoing messages (delivered by applications)
- When done send ACK to coordinator
- When all done coordinator sends CHKP-DONE

An improvement is *incremental snapshot* where only request checkpoints to processes that depend on recovery of the coordinator, i.e., those processes that have received a message causally dependent (directly and indirectly) from one sent by the coordinator after the last checkpoint.

Another way could be the one using a global snapshot algorithm used to take a coordinated snapshot.

Logging

We can start from a checkpoint and replay the actions that are stored in a log file. The method works if the system is piecewise deterministic: execution proceeds deterministically between the receipt of two messages.

Each message's header contains all necessary information to replay the messages and a message is *stable* if it can no longer be lost.
For each *unstable message m* define:

- *DEP(m)*: Processes that depend on the delivery of *m*, i.e., those to which *m* has been delivered or to which *m'* dependent on *m* has been delivered.
- *COPY(m)*: Processes that have a copy of *m*, but not yet in stable storage. They are those processes that hand over a copy of *m* that could be used to replay it.

In this way we can also characterize an *orphan process* in the following way:

- Let *Q* be one of the surviving processes after one or more crashes
- *Q* is *orphan* if there exists *m* such that *Q* is in *DEP(m)* and all processes in *COPY(m)* have crashed. There is no way to replay *m* since it has been lost
- If each process in *COPY(m)* has crashed, then no surviving processes must be left in *DEP(m)*. This can be obtained by having all processes in *DEP(m)* be also in *COPY(m)*. When a process become dependent on a copy of *m*, it keeps *m*.

Pessimistic Logging: ensure that any unstable message *m* is delivered to at most one process. The receiver will always be in *COPY(m)*, unless it discards the message, and he cannot send any other message until it writes *m* to stable storage.
If communication is assumed to be reliable then logging should ideally be performed before message delivery. (NO orphan processes)

Optimistic Logging: messages are logged asynchronously, with the assumption that they will be logged before any faults occur.
If all processes in *COPY(m)* crash, then all processes in *DEP(m)* are rolled back to a state where they are no longer in *DEP(m)*. (ALLOW orphan processes)

7. AGREEMENT

7.1 Commit protocols

Atomic commit is a form of agreement widely used in database management systems. *The goal is to ensures atomicity* (the "A" in ACID transactions), so we want:

- A transaction either commits or aborts
- If it commits, its updates are durable
- If it aborts, it has no side effects
- Also consistency (preserving invariants) relies on atomicity
- If the transaction updates data on multiple nodes (partitioned / sharded data base), either all nodes commit or all nodes abort, or if any node crashes, all must abort.

We have two protocols:

1. 2PC – 2 Phase Commit
2. 3PC – 3 Phase Commit

2 Phase Commit (2PC)

In this protocol we have, for sure, a coordinator (transaction manager) and some participants. It works in the following way: a client begins a transaction T sending the message to the participants, execute T and then send a `commit` message to the coordinator. He sends a `prepare` message to the participants and they decide to commit or to abort, then there is the final message that is what they have decided.

The coordinator and participants can be in 4 states:

1. Init
2. Wait – Ready (for participant)
3. Abort
4. Commit

For the coordinator: Init → Wait → Abort or Commit

For the participant: Init → Abort (directly) or Ready → Commit or Abort (if you were in Ready)

What happens if there is a failure?

- A participant fails – after a timeout, the coordinator assumes abort.
- The coordinator fails
    - Participant waiting for prepare (Init state) → it can decide to abort (no participant can have already received a global commit decision)
    - Participant waiting for global decision (Ready state) → it cannot decide on its own. It can wait for the coordinator to recover, or it can request the decision to another participant, which
        - May have received a reply from the coordinator
        - May be in Init state → coordinator has crashed before completing the prepare phase → assume abort

    - What if everybody is in the same Ready situation? Nothing can be decided until the coordinator recovers. **Blocking protocol!**

2PC is **safe** (never leads to an incorrect state) but it may block. We assumed all nodes (including the coordinator) need to reach an agreement. In that case, 2PC is vulnerable to a single-node failure (the coordinator). If it takes time to restore the failed node (e.g., manual procedure), the system remains unavailable.

3 Phase Commit (3PC)

Like the *2 Phase Commit protocol* but trying to solve the problems by adding another phase between the Ready state and the Commit/Abort one.

For the coordinator: Init → Wait → Abort or Pre-Commit → Commit (if you were in Pre-Commit)

For the participant: Init → Abort (directly) or Ready → Pre-Commit or Abort (if you were in Ready) → Commit (if you were in Pre-Commit)

What happens if there is a failure?

- A participant fails
    - Coordinator blocked waiting for vote (Wait state) can assume abort decision
    - Coordinator blocked in state Pre-commit can safely commit and tell the failed participant to commit when it recovers
- The coordinator fails
    - Participant blocked waiting for prepare (Init state) can decide to abort
    - Participant blocked waiting for global decision (Ready state) can contact another participant:
        - Abort (at least one) → Abort
        - Commit (at least one) → Commit
        - Init (at least one) → Abort
        - Pre-commit (at least one), #Pre-commit + #Ready form a majority → Commit
        - Ready (majority), no-one in Pre-commit → Abort
    - Alternatively, it is possible to elect new coordinator
    - No two participants can be one in Pre-commit and the other in Init

This protocol does not work in case of network partitions.

3PC (quorum-based version presented above) guarantees **safety**: it never leads to an incorrect state.
In a *synchronous system*, it is also guaranteeing *liveness*: it never blocks if a majority of nodes are alive and can communicate with each other (we can use a timeout to learn if a node is connected or not).
In an *asynchronous system*, the protocol *may not terminate* (intuitively, we cannot use finite timeouts to discriminate connected and disconnected nodes.
More expensive than 2PC.

CAP theorem

Any distributed system where nodes share some (replicated) shared data can have at most two of these three desirable properties:

- C: consistency equivalent to have a single up-to-date copy of the data
- A: high availability of the data for updates (liveness)
- P: tolerance to network partitions

In presence of network partitions, one cannot have perfect availability and consistency. Modern data systems provide suitable balance of availability and consistency for the application at hand.

7.2 Replicated state machine – Paxos / Raft

We need a consensus algorithm because it allows collection of machines (servers) to work as coherent group and clients see them as a single (fault-tolerant) machine.
A replicated log ensures that state machines execute the same command in the same order.

This algorithm must work with asynchronous communication (unreliable) and the processes may fail and must remember what they were doing.

It has also ensured

- *safety*, all non-failing machines execute the same command in the same order, and
- *liveness / availability*, the system makes progress if any majority of machines are up and can communicate with each other.

One algorithm was **Paxos,** but it had few problems; it was very difficult to understand, and it allowed only agreement on a single decision, not on a sequence of requests.

Raft
It is equivalent to multi-Paxos. The goal is the *understandability*, and the approach is the *problem decomposition*.

3 key points:

- Leader election
    - Select one server to act as leader
    - Detect crashes and choose a new leader
- Log replication
    - Leader accepts commands from client and append them to its log
    - Leader replicates its log to other servers
- Safety (keep log consistent)
    - Only servers with up-to-date logs can become leader

Nodes can be in three states:

1. Follower (all nodes start as follower)
2. Leader
3. Candidate

All commands go through the leader, who is responsible for committing and propagating them.

Terms

Raft divides time into *terms* of arbitrary length:

- Terms are numbered with consecutive integers
- Each server maintains a *current term* value exchanged in every communication
- Terms identify obsolete information
- Each term begins with an election, in which one or more candidates try to become leader
- There is at most one leader per term.

Leader election

If followers don't hear from a leader for a while, they become *candidate*. A candidate runs an election: if it wins, it becomes the leader.
Followers have a timeout: if they don't hear from the leader, they start an election. Timeout is randomized to avoid many parallel elections and it is in the range: 150 – 300ms.

Who becomes the leader?
Who receives more votes.

The election guarantees:

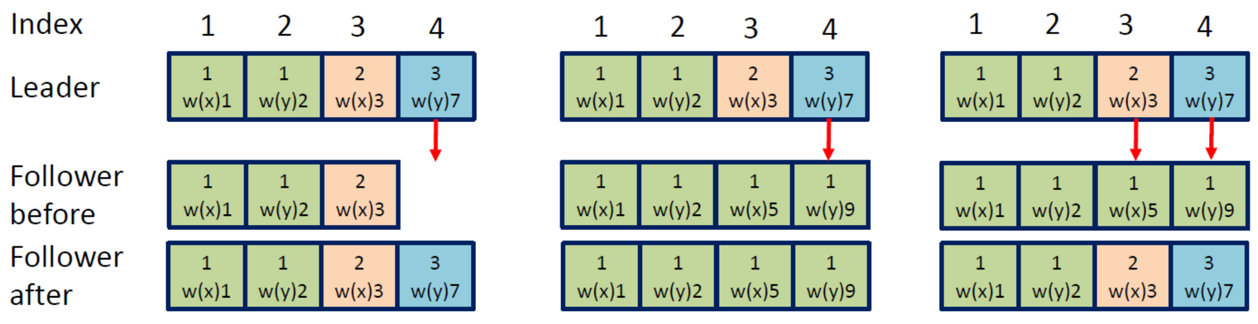
- *safety* because it allows at most one winner per term and some candidate must win. (it works well if timeout time >> communication time)
- *liveness* because some candidate must eventually win. One server usually time out and wins election before others time out.

Log

In a normal situation it is the leader the responsible for writing on the log.
Crashes can result in log inconsistencies, but leader assumes its log is correct and normal operation will repair them.

Raft also guarantees the *log matching property* in fact, if log entries on different servers have the same index and term, they store the same command, and the logs are identical in all preceding entries. If a given entry is committed, all preceding entries are also committed.

There is of course a consistency check on the log.
"*AppendEntries*" messages include `<index, term>` of the entry preceding the new one(s).
Follower must contain matching entries otherwise it rejects the request, and the leader retries with lower log index.


Index
1 2 3 4
Leader
1 w(x)1 | 1 w(y)2 | 2 w(x)3 | 3 w(y)7
Follower before
1 w(x)1 | 1 w(y)2 | 2 w(x)3
1 w(x)1 | 1 w(y)2 | 1 w(x)5 | 1 w(y)9
Follower after
1 w(x)1 | 1 w(y)2 | 2 w(x)3 | 3 w(y)7
1 w(x)1 | 1 w(y)2 | 1 w(x)5 | 1 w(y)9
1 w(x)1 | 1 w(y)2 | 2 w(x)3 | 3 w(y)7

Example 1: ok Example 2: mismatch Example 3: ok

Once log entry committed, all future leaders must store that entry.
Servers with incomplete logs must not get elected. Candidates include index and term of last log entry in `RequestVote` messages. (Voting server denies vote if its log is more up to date)

Communication

In Raft, clients always interact with the leader. When a client starts, it connects to a random server, which communicates to the client the leader for the current term. If a leader crashes, the communication with the client times out.

Raft guarantees *linearizable semantics*; all operations behave as if they were executed once and only once on a single copy.

In the case of a *leader failure*, a client may need to retry. To do so, it attaches a sequential identifier of the request, the servers store the last identifier for each client as part of the log and so they can discard duplicates.

Use in distributed DBMS

Some modern DBMS integrate replicated state machines and commit protocols because the database is partitioned, and each partition is replicated.

- Replicated state machine to guarantee that individual partitions do not fail. Each partition is not managed by a single machine but by a set of machines that behave as one.
- 2PC executes atomic commit across partitions. Coordinator (transaction manager) and participants (partitions) are assumed not to fail as they are replicated; channels are assumed to be reliable.

7.3 Byzantine conditions

State machine replication can be extended to consider byzantine processes known as Byzantine Fault Tolerant (BFT) replication. Same assumption as Paxos/Raft and requires ***3f+1*** participants to tolerate ***f*** failing nodes.

Cryptocurrencies can be seen as replicated state machines where the state is the current balance of each user, and everything (transactions/payments) is stored in a replicated ledger (log). Byzantine situation when a user may try to "double spend" their money or may try to create inconsistent copies of the log.

Bitcoin blockchain

The blockchain is the public ledger that records transactions, it contains all transactions from the beginning of the blockchain and each block of the chain includes multiple transactions.

It is a distributed ledger (replicated log) and transactions are digitally signed; if you lose the private key, bitcoins are lost too.

Adding a block to the chain requires solving a mathematical problem (proof of work). The solution of this problem is difficult to find, and the complexity is adapted to computational power, but it is easy to verify the validity of the new block created.

Proof of work computed by special nodes (miners) that collect new (pending) transactions into a block (they earn Bitcoins if successful). When a miner finds the proof, it broadcasts the new block.

The other miners receive it and try to create the next block in the chain (global agreement on the order of blocks).

What if two miners find a proof concurrently? Very unlikely that two computers will find a solution at the same time.
If two concurrent versions are created, the one that grows faster (includes more blocks) survive.

8. REPLICATION and CONSISTENCY

8.1 Introduction

Why we need “replication”?

- To achieve fault tolerance
- To increase availability
- To improve performance (sharing of workload to increase the *throughput* of served requests and reduce the *latency* for individual requests)

The main problem of replication is *consistency* across replicas. Also, scalability is a problem because replication may degrade performance.

A consistency model is a contract between the processes and the data store.
Several models offer different guarantees:

- Guarantees on content – Maximum “difference” on the versions stored at different replicas
- Guarantees on staleness – Maximum time between a change and its propagation to all replicas
- Guarantees on the order of updates – Constrain the possible behaviours in the case of conflicts

Consistency protocols

- **Single leader protocol**
  - One of the replicas is designated as the leader
    - Clients want to write; they must send a request to the leader which first writes the new data to its local storage
  - The other replicas are known as followers
    - Whenever the leader writes new data to its local storage, it also sends the data to all its followers
  - When a client wants to read from the database:
    - In some systems it queries the leader
    - In other systems it can query any replica
  - Can be
    - Synchronous – The write operation completes after the leader has received a reply from all the followers
    - Asynchronous – The write operation completes when the new value is stored on the leader and followers are updated asynchronously
    - Semi-synchronous – The write operation completes when the leader has received a reply from at least *k* replicas
  - Synchronous (or semi-synchronous with k followers) replication is safer because even if *k-1* replicas fail, we still have a copy of the data and followers can recover from failures by asking updates to other replicas (catch-up recovery)
  - What happens if the leader fails?
    - Elect a new one
    - To ensure safety, we need some consensus protocol

    - This protocol is adopted in distributed databases like PostgreSQL, MySQL, Oracle, SQL Server, and MongoDB in single organizations / data center.
    - No write-write conflicts possible
        - The leader receives all write operations and determines their order
    - Read-write conflicts still possible
        - Depending on the specific implementation, for example, a client reads from an asynchronous replica and does not see writes it previously performed on the leader
- **Multi leader protocol**
    - No single leader means that there is no single entity that decides the order of writes
    - Writes are carried out at different replicas concurrently
    - It is possible to have write-write conflicts in which two clients update the same value almost concurrently
    - It is often adopted in geo-replicated settings, and it is more difficult to handle but, in practice, conflicts are rare and easy to solve
- **Leaderless protocol**
    - The client contacts *multiple* replicas to perform the writes/reads. In some implementations, a coordinator forwards operations to replicas on behalf of the client
    - Leaderless replication uses *quorum-based protocols* to avoid conflicts
        - We need a majority of replicas to agree on the write
        - We need an agreement on the value to read

We will distinguish models that can be implemented with highly available protocols and models that cannot. In this context, we say that a protocol is *highly available* if it does not require synchronous/blocking communication. If a node or a network link fails, a client can still receive a reply from a correct (non-failed) replica.

High availability is related to the CAP theorem:

- C = consistency
- A = availability
- P = network partition (failures)

In the presence of network failures (P), you can have availability (A) or consistency (C), but not both.

8.2 Data-centric consistency model

Ideally, we would like all operations to take place instantaneously at some point in time and be globally ordered according to their time of occurrence.

Sequential consistency (sequential order)

- Operations within a process may not be re-ordered
- Does not rely on time

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)b	R(x)a

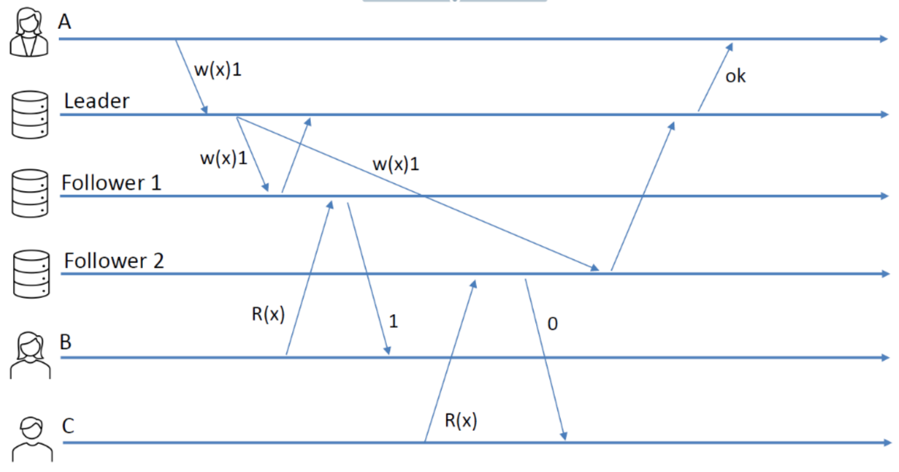
Consistent

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

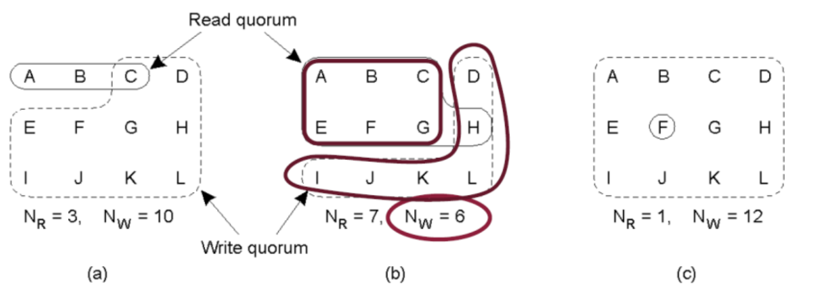
NOT Consistent

- All the replicas need to agree on a given order of operations
  - Single leader replication with synchronous replication
    - Links are FIFO and clients are "*sticky*"
  - Leaderless implementation – Quorum based
    - Clients contact multiple replicas to perform a read or a write operation
    - An update occurs only if a quorum of the servers agrees on the version number to be assigned
    - Reading requires a quorum to ensure the latest version is being read
    - NR + NW > N
    - NW > N/2

Single leader implementation



Leaderless implementation



A correct choice of read and write set

A choice that may lead to write-write conflicts

A correct choice, known as ROWA (read one, write all)

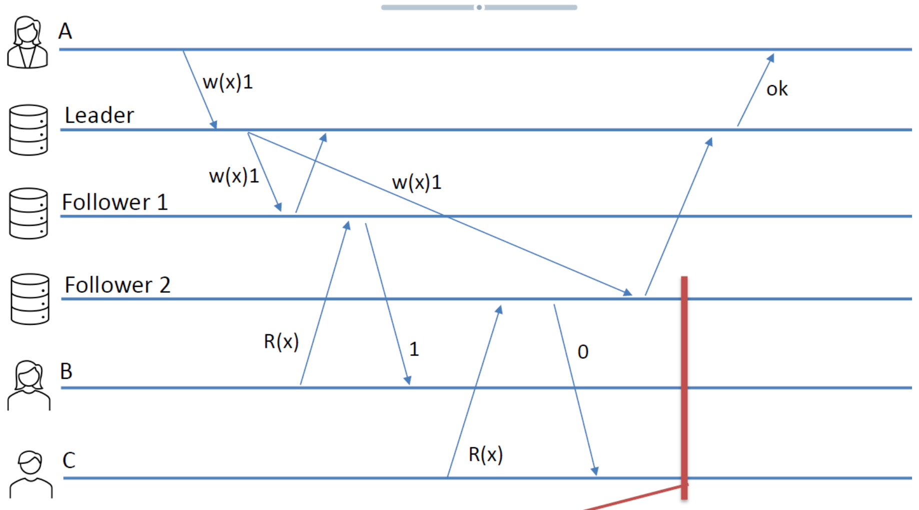
- Two main problems related to availability

- High latency due to synchronous interactions
- Clients are blocked in the case of network partitions

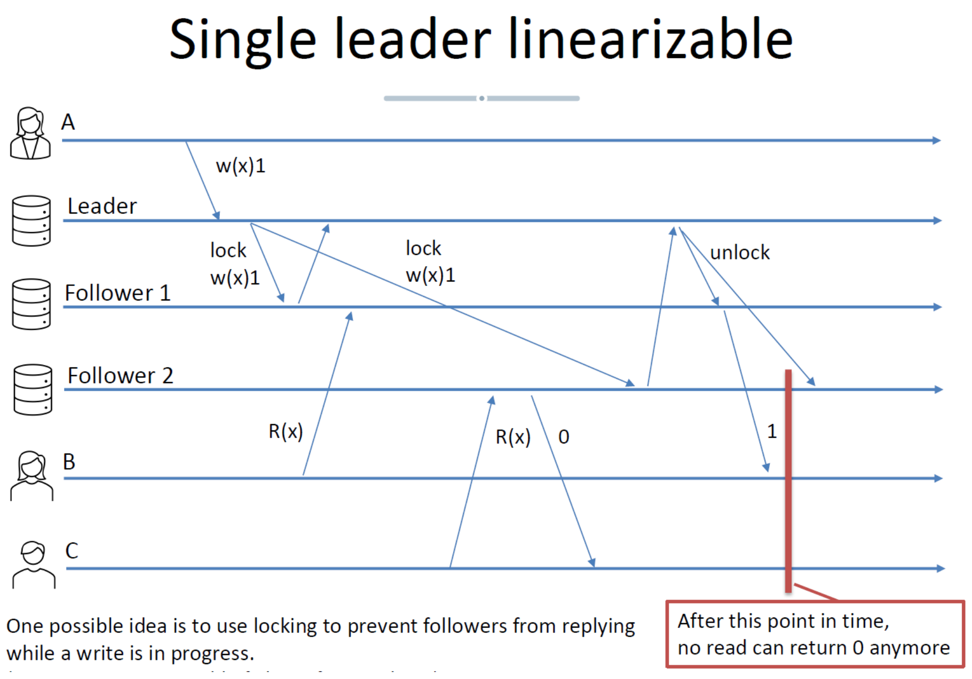
Linearizability

- Strongest possible consistency guarantees in presence of replication
- When a client completes a write on the data-store, all clients need to see the effect of the write
- It includes a notion of time: operations behave as if they took place at a single point of (wall clock) time
- Operations have a duration
- Linearizability is a *composable* property; if the operations on individual variables are linearizable, also the global schedule is also linearizable
  - Single leader implementation
    - The leader orders the writes according to their timestamp
    - Replicas are updated synchronously and atomically

Linearizability



The schedule is sequential, but …
At this point in time B thinks x is already 1, C thinks x is still 0
If they communicate (through a different channel), they break the illusion of a single copyç

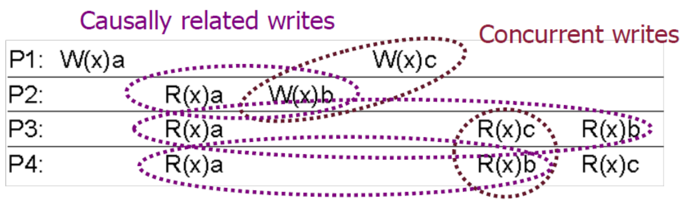


Single leader linearizable

One possible idea is to use locking to prevent followers from replying while a write is in progress.

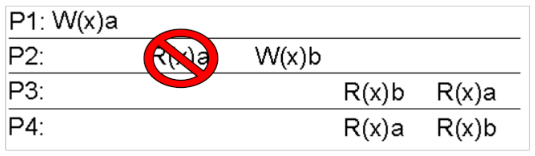
Casual consistency

- Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in any order at different machines



P1:	W(x)a		W(x)c	
P2:	R(x)a	W(x)b		
P3:	R(x)a		R(x)c	R(x)b
P4:	R(x)a		R(x)b	R(x)c

Consistent



P1: W(x)a				
P2:	~~R(x)a~~	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

NOT Consistent
(it becomes consistent without P2's R(x))

- Causal consistency defines a causal order among operations
  - A write operation *W* by a process *P* is causally ordered after every previous operation *O* by the same process
  - A read operation by a process *P* on a variable *x* is causally ordered after a previous write by *P* on the same variable
  - Causal order is transitive
- It is not a total order. Operations that are not causally ordered are said to be concurrent

Casual consistency is easier to guarantee within a distributed environment and it is also easier to implement.

- Multi leader implementation
  - Writes are timestamped with vector clocks
  - Vector clocks define what the process knew when it performed the write

    - An update U is applied to a replica only when all the write operations that are possible causes of U have been received and applied
    - Highly available
    - This implementation works if the clients are sticky

FIFO consistency – Also called PRAM consistency (Pipelined RAM)

- If writes are put onto a pipeline for completion, a process can fill the pipeline with writes, not waiting for early ones to complete
- Very easy to implement
- A replica performs an update U from P with sequence number S only after receiving all the updates from P with sequence number lower than S
- It still requires all writes to be visible to all processes, even those that do not care

Consistency	Description
Linearizable	All processes must see all shared accesses in the same order. Operations behave as if they took place at some point in (wall-clock) time.
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time.
Casual	All processes see causally related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.

Eventual consistency

It does not guarantee any order, but it guarantees that the final destination is the same. It does not tell anything about the order of the operations.

Eventual consistency **does not imply** FIFO, sequential or causal consistency.

- Updates are guaranteed to eventually propagate to all replicas
- Used in web caches, DNS, social media (geo distributed data stores)
- Very easy to implement
- Very few conflicts in practice
- Reasonable trade-off between performance and complexity
- *Conflict-free replicated data types* (CRDTs) guarantee convergence even if updates are received in different orders.
  Limitations: they are not useful for general data structure.

8.3 Client-centric consistency model

Client-centric consistency models that provide guarantees about accesses to the data store from the perspective of a single client.
Client-centric consistency models are relevant in distributed systems, where data is stored and processed across multiple nodes or servers. These models aim to ensure consistency in the data observed by clients interacting with the system.

Monotonic reads: if a process reads the value of a data item *x*, any successive read operation on *x* by that process will always return that same value or a more recent value.

Monotonic writes: a write operation by a process on a data item *x* is completed before any successive write operation on *x* by the same process.

Read your writes: the effect of a write operation by a process on a data item *x* will always be seen by a successive read operation on *x* by the same process.

Writes follow reads: a write operation by a process on a data item *x* following a previous read operation on *x* by the same process is guaranteed to take place on the same or more recent value of *x* that was read.

Implementation:

- Each operation gets a unique identifier (replica id + sequence number)
- Two sets are defined for each client
    - Read-set: the write identifiers relevant for the read operations performed by the client
    - Write-set: the identifiers of the write performed by the client
- Can be encoded as vector clocks
    - Latest read/write identifier from each replica

8.4 Design strategies

How to place replicas?

- Permanent replicas statically configured like CDN or DNS
- Server-initiated replicas (created dynamically to cope with access load or to move data closer to clients)
- Client-initiated replicas that rely on client caches

How to propagate?

- Push-based approach – the update is propagated to all replicas, regardless of their needs
- Pull-based approach – an update is fetched on demand when needed (more convenient when #reads << #writes and it is used to manage client caches)
- Leases can be used to switch between the two

Propagation strategies?

- Leader-based protocols
- Leaderless protocols
    - Read repair – when a client makes a read from several replicas in parallel, it can detect stale responses from some replica. The client (or a coordinator on its behalf) updates the state replica
    - Anti-entropy process – background process that constantly checks for differences among replicas and copies missing data

9. BIG DATA PLATFORMS

9.1 Introduction

Data science is an interdisciplinary field that uses scientific methods, processes, algorithms, and systems to extract knowledge and insights from data in various forms, both structured and unstructured.

- Collect all the data – the more the better → statistical relevance
- Decide what to do with data
- There is a huge difference with respect to traditional information systems

As consequences, we have:

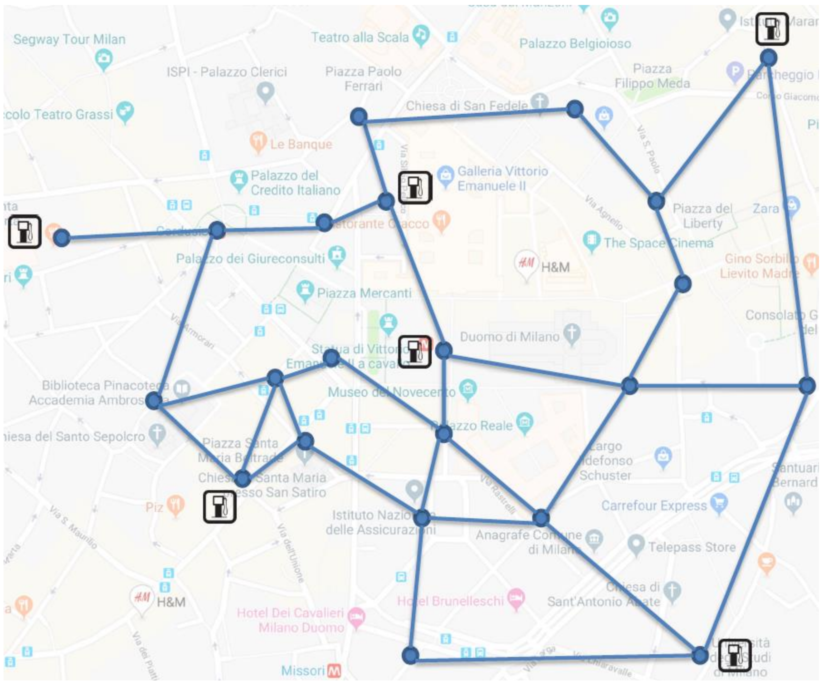
- Volume – the volume of data is a lot
- Velocity – receive a lot of data rapidly
- Variety – data in different formats
- Veracity – not always correct data

Problems of big data:

- Scalability to large data volumes
- Support for quickly changing data
- Require functions like automatic parallelization & distribution, monitoring tools, fault-tolerance, etc.

9.2 MapReduce

We have a map represented as a graph where nodes are POI and edges are roads and we want to compute the shortest path from each point to the nearest gas station.



We can compute the shortest path between each gas station and each node using well known algorithm like Dijkstra and, then, for each node, we need to compare the distance computed from each gas station and keep only the minimum.
As optimization, we can discard nodes that are too far away, e.g., assuming that the max distance between any node and the nearest gas station is 50 km, we can discard paths that are longer than that.

This is a small problem, but, in general, some problems only appear "*at scale*". Reading input and reading/storing the intermediate state of the computation can easily become the bottleneck, or if input/state do not fit in one machine, we need distributed solutions and so on.

So, we need to consider the scale of the problem and the computing infrastructure we have.

Now, imagine having a datacentre, we can split the dataset into blocks.

- Each block b_i centred around gas station *i*
- Different blocks can be stored on different computers
- For each block b_i, we can compute the distance between gas station *i* and any node in the block
    - The computation is independent for each block
    - Blocks can be processed entirely in parallel

Next, we need to determine the closest gas station and the shortest path for each node. This requires communication because a point can be at the intersection of several blocks. We can repartition the data by node.
(the shortest path can be computed independently for each node)

This is an example of **MapReduce**.

2 phases: Map and Reduce

1. Map processes individual elements – for each of them outputs one or more <key, value> pairs. (sorting and filtering data)
2. Reduce processes all the values with the same key and outputs a value.

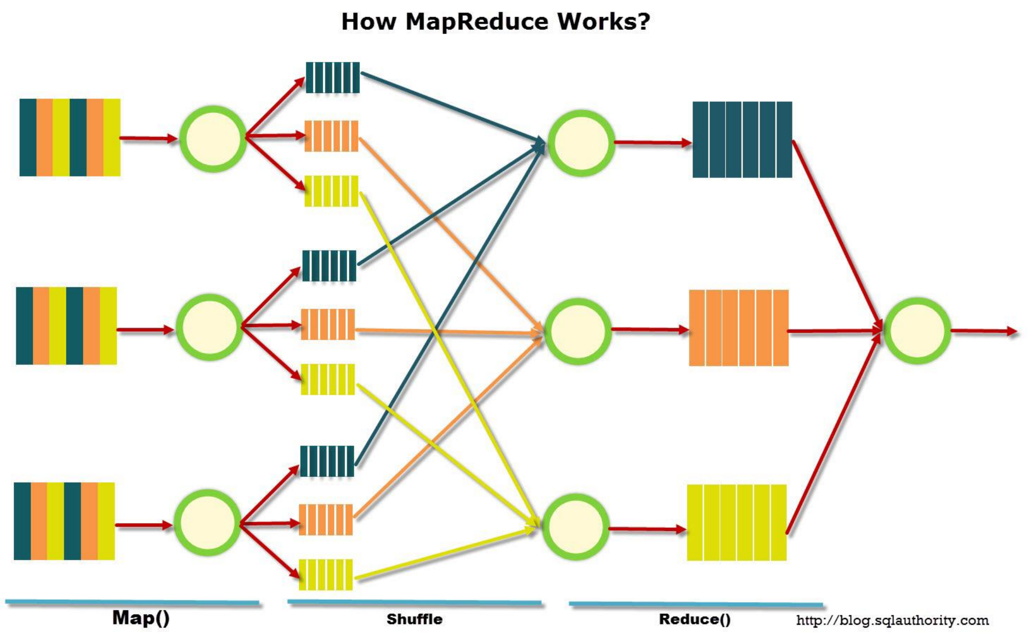
The **key principles** of MapReduce are *fault tolerance*, *scalability*, and *the ability to handle large datasets* by distributing the workload across multiple nodes. It allows computations to be carried out closer to the data, reducing the need for data movement across the network and significantly improving performance.

What does the platform do?

- Scheduling: allocates resources for mappers and reducers.
- Data distribution: moves data from mappers to reducers.
- Fault tolerance: transparently handles the crash of one or more nodes.
    - On worker failure
        - Master detects failure via periodic heartbeats
        - Both completed and in-progress map tasks on that worker should be re-executed (Output stored on local disk)

        - Only in-progress reduce tasks on that worker should be re-executed (Output stored in global file system)
        - All reduce workers will be notified about any map re-executions
    - On master failure – State is check-pointed to GFS (Google File System): new master recovers & continues

Stragglers are tasks that take long time to execute due to slow hardware, poor partitioning, etc.



- PROs
    - The developers write simple functions
    - The system manages complexities of allocation, synchronization, communication, fault tolerance, stragglers, …
    - Very general
    - Good for large-scale data analysis
- CONs
    - High overhead
    - Lower raw performance than HPC (High Performance Computing)
    - Very fixed paradigm (each MapReduce step must complete before the next one can start)

9.3 Beyond MapReduce

Many systems extended and improved the MapReduce abstraction in many ways.

We will use a dataflow model that is implemented into two different architectures:

- Scheduling of tasks – Apache Spark (batch or micro-batch processing)
    - The dataset is always the same, never change
    - Similar to MapReduce but there are arbitrary number of stages
    - Intermediate results can be cached in main memory if they are reused multiple times
    - It has also support for streaming data

        - The data is split into small batches
- Pipelining of tasks – Apache Flink (streaming or continuous processing)
    - All the operators are instantiated as soon as the job is submitted
    - They communicate using TCP channels
    - An operator can start processing as soon as it has some data available from the previous ones. Pipeline architecture where multiple operators are simultaneously running.
    - Low latency since the data is processed immediately

COMPARISON		
	Spark	**Flink**
Latency	Higher latency	Lower latency (process data immediately)
Throughput	Moving larger data blocks can be more efficient	
Load balancing	Dynamic scheduling decisions can consider data distribution	The load could not be balanced. Allocation of tasks to operators is decided statically when the job is deployed
Elasticity	Scheduling decisions take place dynamically at runtime. It is possible to dynamically change the set of physical resources being used for deployment	Elasticity is simply not possible in pipelined approaches.
Fault-tolerance	What happens if a node fails? Re-schedule the task. If a data element is lost simply, recompute it. If the data depends on is lost simply recompute it	Periodically checkpoint to (distributed) file system In the case of failure, replay from the last checkpoint

10. P2P

10.1 Introduction

Alternative paradigm that promotes the sharing of resources and services through direct exchange between peers. Some examples: network bandwidth (Internet), processing cycles (Bitcoin), storage space (Freenet), data (most of the rest).

All nodes are potential users of a service and potential providers of a service. Nodes act as servers, clients, as well as routers and each node is independent of the other (no central administration) and they are dynamic (they come and go unpredictably).

The scale of the system can be Internet-wide, in fact, we do not have a global view of the system and resources are geographically distributed.
(*Overlay network* – e.g., three nodes seem close in an overlay network but they geographically far away each other)

Retrieving resources is a fundamental issue in P2P systems due to their inherent geographical distribution. We can distinguish two forms of retrieval operations that can be performed on a data repository:

1. Search for something
2. Lookup a specific item

But what to retrieve?

- The actual data only if we performed a lookup operation
- A reference to the location from where the data can be retrieved (in both cases, lookup or search)

10.2 Napster (Centralized search)

It was the first p2p file sharing application. The idea was to share the storage and bandwidth of individual users (home users).

It uses a centralized search because it relies on a central server used as a search engine.
Clients connect to a central server; send to it a list of files they have on their pcs or they can search for someone owning a specific file and get it directly from the peer.

Since it has a centralized server and the whole system depends on it, it is not a "*pure*" p2p architecture.

- PROs:
  - Simple and easy to search a file O(1)
- CONs:
  - Single point of failure
  - Single point of control
  - Server processes all the requests

10.3 Gnutella (Query flooding)

Gnutella is a decentralized peer-to-peer (P2P) file-sharing protocol used for sharing files over the internet.

In Gnutella's P2P network, participants (peers) can both share and download files directly with each other without relying on a central server. This decentralized structure means that no single entity has control over the entire network. Users can search for files across the network and download them from other users who are sharing those files. Additionally, users can share their own files, contributing to the collective availability of content within the network.

Each query is forwarded to all neighbours and the propagation is limited by a *HopsToLive* field in the messages.

Joining the network:

- The new node connects to a well-known "*anchor*" node
- Then sends a PING message to discover other nodes
- PONG messages are sent in reply from hosts offering new connections with the new node
- Direct connections are then made to the newly discovered nodes

PROs:

- Fully decentralized (no central coordination required)
- Search cost distributed
- "*Search for S*" can be done in many ways (structured database search, simple text matching, "fuzzy" text matching, etc.)

CONs:

- "*Flood*" of Requests. If average number of neighbours is C and average HTL is D, each search can cause CxD request messages
- Search scope is O(N)
- Search time is O(2D)
- Nodes leave often, so network unstable

10.4 KaZaA (Hierarchical topology)

Kazaa was a peer-to-peer file-sharing application. It allowed users to share and download files, including music, videos, software, and documents, directly from other users' computers connected to the Kazaa network.

There is a distinction in the network between "*normal nodes*" and "*supernodes*".
Normal nodes contact supernodes to perform searches and they are flooded by queries. So, client contacts a supernode and sends a list of files or ask for something, then the supernode floods query amongst the others and get the file directly from peer(s).

PROs:

- Tries to consider node heterogeneity
- KaZaA rumoured to consider network locality

CONs:

- Still no real guarantees on search scope or search time

10.5 BitTorrent (Collaborative systems)

It allows many people to download the same file without slowing down everyone else's download. It does this by having downloaders swap portions of a file with one another, instead of all downloading from a single server. Such contributions are encouraged because every client trying to upload to other clients gets the fastest downloads.

- Join: contact centralized "*tracker*" server, get a list of peers
- Publish: run a tracker server
- Search: out-of-band (search on Google what you want)
- Fetch: download chunks of the file from your peers.

Terminology:

- Torrent: a meta-data file describing the file(s) to be shared
- Seed: a peer that has the complete file and still offers it for upload
- Leech: a peer that has incomplete download
- Swarm: all seeds/leeches together make a swarm
- Tracker: a server that keeps track of seeds and peers in the swarm and gathers statistics

How is the content distribution done?

- The file is broken into smaller fragments, usually 256KB.
- Peers contact the tracker to have a list of the peers.
- Peers download missing fragments from each other and upload to those who don't have it.
- The fragments are not downloaded in sequential order and need to be assembled by the receiving machine. When a client needs to choose which segment to request first, it usually adopts a "*rarest-first*" approach, by identifying the fragment held by the fewest of its peers. This tends to keep the number of sources for each segment as high as possible.
- Clients start uploading what they already have (small fragments) before the whole download is finished.
- Once a peer finish downloading the whole file, it should keep the upload running and become an additional seed in the network.
- Everyone can eventually get the complete file as long as there is "one distributed copy" of the file in the network, even if there are no seeds.

How are peers chosen?
The decision on which peers to un/choke is based solely on download rate, which is evaluated on a rolling, 20-second average.
The selected peer rotates every 30s, this allows to discover currently unused connections that are better than the ones being used.

PROs:

- Works reasonably well in practice

- Gives peers incentive to share resources; avoids free-riders

CONs:

- Pareto efficiency is a relatively weak condition
    - If two peers get poor download rates for the uploads they are providing, they can start uploading to each other and get better download rates than before
- Central tracker server needed to bootstrap swarm

10.6 Freenet (Secure storage)

Freenet is a P2P application designed to ensure true freedom of communication over the Internet. It allows anybody to publish and read information with reasonable anonymity. Rely on no centralized control or network administration, it is scalable from tens to hundreds of thousands of users and is robust against node failure or malicious attack.

- Join: clients contact a few other nodes they know about; get a unique node id
- Publish: route file contents toward the node which stores other files whose id is closest to file id
- Search: route query for file id using a steepest-ascent hill-climbing search with backtracking
- Fetch: when query reaches a node containing file id, it returns the file to the sender

How does the routing protocol work?
Each node in the network stores some information locally and they also have approximate knowledge of what their neighbours store too.
Request is forwarded to node's "*best guess*" neighbour unless it has the information locally. If the information is found within the request's "*hops to live*", it is passed back through this chain of nodes to the original requestor. The intermediate nodes store the information in their LRU (least recently used) cache as it passes through.

How does the publication work?
Inserted data is routed in the same way as a request would.

- Search for the *id* of the data to insert
- If the *id* is found, the data is not reinserted (it was already present). Otherwise, the data is sent along the same path as the request. (this ensures that data is inserted into the network in the same place as requests will look for it)

During searches data is cached along the way and each node adds entry to routing table associating the key and the data source.

Routing properties:

- "*close*" file ids tend to be stored on the same node.
- Networks tend to be a "*small world*". Most nodes have few local connections, but a few nodes have a lot of neighbours, so well-known nodes tend to see more requests and become even better connected and most queries only traverse a small number of hops to find the file.

Caching properties:

- Information will tend to migrate towards areas of demand
- Popular information will be more widely cached
- Files prioritized according to popularity
- Unrequested information may be lost from the network

Anonymity & security:

- Nodes cannot tell where a message originated
- The ids of two files should not collide
- Use a robust hashing so that the id of a file is directly related to its content
- Link-level encryption
- Document encryption and verification

PROs:

- Intelligent routing makes queries relatively short
- Search scope small
- Only nodes along search path involved
- No flooding
- Anonymity properties may give you "*plausible deniability*"

CONs:

- Still no provable guarantees!
- Anonymity features make it hard to measure, debug

10.7 Distributed Hash Tables (Structured topology)

A distributed "*hash-table*" (DHT) data structure; put(id, item) & item = get(id).

- Join: clients contact a "*bootstrap*" node and integrate into the distributed data structure; get a node id
- Publish: route publication for file id toward a close node id along the data structure
- Search: route a query for file id toward a close node id. The data structure guarantees that query will meet the publication
- Fetch:
    - Publication contains actual file → fetch from where query stops
    - Publication contains a reference → fetch from the location indicated in the reference

An example of DHT is *Chord*

- Nodes and keys are organized in a *logical ring*
- Each node is assigned a unique m-bit identifier
- Every item is assigned a unique m-bit key (the hash of the item)
- The item with key k is managed by the node with the smallest $id \geq k$ (the successor)
    - Basic lookup

        - Each node keeps track of its successor
        - Find a node asking to successor if it manages that key
    - Finger table
        - Each node maintains a finger table with m entries
        - Entry *i* in the finger table of node *n* is the first node whose id is higher or equal than $n + 2^i$
- Joining a DHT – Chord
    - When a new node *n* joins, the following actions must be performed
        - Initialize the predecessor and fingers of node *n*
        - Update the fingers and predecessors of existing nodes to reflect the addition of *n*
        - To initialize its predecessor and fingers, we assume n knows another node n' already into the system
            - It uses *n'* to initialize its fingers
            - Finger *i* = successor of node $\mathrm{n} + 2^i$
        - To update fingers of other nodes, we observe that node *n* will become the *i-th* finger of node *p* if and only if
            - *p* precedes *n* by at least 2^{i-1}
            - The *i-th* finger of *p* succeeds *n*
        - The first node *p* that meets these two conditions is the immediate predecessor of node $n - 2^{i-1}$
            - Search for this node and update it

The correctness of Chord relies on the correctness of successors pointers, in fact, to stabilize routing information, Chord uses periodic procedures to update successor and finger tables.

To increase robustness, each Chord node maintains a list of successors of size *R*, so, when the immediate successor does not respond, it can contact the next in the list.
(Resilient even if *R-1* successors fail simultaneously)

- Routing table size?
    - $\log n$ fingers
    - With $n = 2m$ nodes
- Routing time?
    - Each hop expects to $\frac{1}{2}$ the distance to the desired id => expect $O(\log n)$ hops
- Joining time?
    - With high probability expect $O(\log_2 n)$ hops

PROs:

- Guaranteed Lookup
- $O(\log n)$ per node state and search scope

CONs:

- No one uses them? (only one file sharing app)
- It is more fragile than unstructured networks

- Supporting non-exact match search is hard

It does not consider physical topology.